DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
COLLEGE OF ENGINEERING & TECHNOLOGY
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

STRATEGIES FOR CONCURRENT PROCESSING OF COMPLEX ALGORITHMS IN DATA DRIVEN ARCHITECTURES

By

John W. Stoughton, Principal Investigator

Roland R. Mielke, Co-Principal Investigator

Sukhamoy Som, Graduate Research Assistant

Rodrigo Obando, Graduate Research Assistant

Robert Tymchyshyn, Graduate Research Assistant

Progress Report
For the period May 16, 1987 to May 15, 1988

Prepared for the
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Under
Research Grant NAG-1-683
Mr. Paul J. Hayes, Technical Monitor
ISD-Information Processing Technology Branch

June 1988

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
COLLEGE OF ENGINEERING & TECHNOLOGY
OLD DOMINION UNIVERSITY
NORFOLK, VIRGINIA 23529

STRATEGIES FOR CONCURRENT PROCESSING OF COMPLEX ALGORITHMS IN DATA DRIVEN ARCHITECTURES

By

John W. Stoughton, Principal Investigator

Roland R. Mielke, Co-Principal Investigator

Sukhamoy Som, Graduate Research Assistant

Rodrigo Obando, Graduate Research Assistant

Robert Tymchyshyn, Graduate Research Assistant

Progress Report
For the period May 16, 1987 to May 15, 1988

Prepared for the
National Aeronautics and Space Administration
Langley Research Center
Hampton, VA 23665

Under
Research Grant NAG-1-683
Mr. Paul J. Hayes, Technical Monitor
ISD-Information Processing Technology Branch

Submitted by the
Old Dominion University Research Foundation
P. O. Box 6369
Norfolk, Virginia 23508

June 1988

DISCLAIMER

The use of brand names in this document is for completeness only and does not imply NASA endorsement.

STRATEGIES FOR CONCURRENT PROCESSING OF COMPLEX ALGORITHMS IN DATA DRIVEN ARCHITECTURES

By

John W. Stoughton[1], Roland R. Mielke[2], Sukhamoy Som[3], Rodrigo Obando[4] and Robert Tymchyshyn[5]

ABSTRACT

The purpose of this report is to document research to develop strategies for concurrent processing of complex algorithms in data driven architectures. The problem domain consists of decision-free algorithms having large-grained, computationally complex primitive operations. Such are often found in signal processing and control applications. The anticipated multiprocessor environment is a data flow architecture containing between two and twenty computing elements. Each computing element is a processor having local program memory, and which communicates with a common global data memory. A new graph theoretic model called ATAMM which establishes rules for relating a decomposed algorithm to its execution in a data flow architecture is presented. The ATAMM model is used to determine strategies to achieve optimum time performance and to develop a system diagnostic software tool. In addition, preliminary work on a new multiprocessor operating system based on the ATAMM specifications is described.

[1] Associate Professor, Department of Electrical & Computer Engineering, Old Dominion University, Norfolk, Virginia 23529.
[2] Professor, Department of Electrical & Computer Engineering, Old Dominion University, Norfolk, Virginia 23529.
[3] Graduate Research Assistant, Department of Electrical & Computer Engineering, Old Dominion University, Norfolk, Virginia 23529.
[4] Graduate Research Assistant, Department of Electrical & Computer Engineering, Old Dominion University, Norfolk, Virginia 23529.
[5] Graduate Research Assistant, Department of Electrical & Computer Engineering, Old Dominion University, Norfolk, Virginia 23529.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

TABLE OF CONTENTS (Continued)

LIST OF FIGURES (Continued)

TABLE OF CONTENTS (Concluded)

LIST OF FIGURES (Concluded)

I.0 INTRODUCTION

The purpose of this report is to document research to develop strategies for concurrent processing of complex algorithms in data driven architectures. The problem domain consists of decision-free algorithms having large-grained, computationally complex primitive operations. The anticipated multiprocessor environment is assumed to contain between two and twenty computing elements for concurrent execution of the various primitive operations. Each computing element or functional unit is a processor having local memory for program storage and temporary input and output data containers. The functional units have a common global data memory, and functional unit activity is coordinated by a graph manager. The global memory and graph manager may be either centralized or distributed. The authors have proposed a new graph theoretic model to provide a basis for establishing rules for relating a decomposed algorithm to its execution in a data flow environment. The model is identified by the acronym ATAMM which represents Algorithm To Architecture Mapping Model. The availability of the ATAMM model ils important because it provides a context in which to investigate algorithm decomposition strategies, it provides a basis for predicting and improving time performance, and it identifies the data flow and control flow required of any data flow architecture which implements the algorithm.

During an earlier grant period, May 16, 1986 to May 15, 1987, the authors formulated the ATAMM model for representing the implementation of a decomposed algorithm in a data flow architecture. In addition, a simulation tool was developed to display data flow and control flow for algorithms operating according to the ATAMM rules. During the present grant period, May 16, 1987 to May 15, 1988, the ATAMM model was used to determine analytically performance bounds for task computational time and system throughput

time. An operating strategy which achieves optimum time performance was developed. In addition, a new diagnostic software tool was developed for use with the simulation tool. The diagnostic tool monitors detailed system operation and displays global system performance indicators and measures. Also, a new multiprocessor operating system based on the ATAMM specifications is being constructed to validate the ATAMM rules and to provide a testbed for further experimentation. It is the purpose of this report to a detailed description of the research performed during the present grant period.

In Section II, a overview of research performed during the period May 16, 1987 to May 15, 1988 is presented. This overview consists of summaries of work to develop strategies for optimum time performance, diagnostic software tools, and a testbed operating system. In Section III, the development of strategies for optimum time performance is described. The new diaganostic software tools are explained and illustrated in Section IV. Recommendations for continuing and future research are briefly outlined in Section V. Two papers describing recent research efforts are included as appendices.

II. RESEARCH OVERVIEW

In this section, a summary of research activity conducted during the period May 16, 1987 through May 15, 1988 is presented. A more detailed description of this work, as well as illustrative examples, is given in the following sections and the appendices.

II.1 Modeling and Performance

The development of a new graph theoretic model for describing data and control flow associated with the execution of large-grained aLgorithms in a special distributed computing environment is presented. The model is identified by the acronym ATAMM which represents Algorithm To Architecture Mapping Model. The purpose of such a model is to provide a basis for establishing rules for relating an algorithm to its execution in a multiprocessor environment. Specifications derived from the model lead directly to the description of a data flow architecture. The availability of the ATAMM model is important for at least three reasons. First, it provides a context in which to investigate algorithm decomposition strategies without the need to specify a specific computer architecture. Second, the model identifies the data flow and control dialog required of any data flow architecture which implements the algorithm. Third, the model provides a basis for calculating analytically performance bounds for computing speed and throughout capacity.

The problem domain of the ATAMM model consists of decision free algorithms with computationally complex primitive operations which are assumed to be implemented in a dedicated data flow environment. The algorithms are such as may be found in (but not limited to) large scale signal processing and control applications. The anticipated multiprocessor environment is

assumed to consiste of two to twenty processing elements for concurrent execution of the various algorithm primitives.

The development of new computer architectures based upon distributed, multiprocessor organizations [1], [2] is motivated mainly by the requirement for increased speed and greater throughput capability in complex signal processing applications [3]. Recent advances in the production of high-density microelectronics [4] has made possible the construction of parallel architectures consisting of identical, special purpose computing elements [5]. A number of models for describing the behavior of algorithms in this setting have been developed [6] - [8]. However, these models represent only the data flow and do not adequately display the complex issues of communication and control flow which must occur in any realization of the model. For this reason, it has been difficult to investigate how to effectively match the decomposition and scheduling of algorithms to the structure and control of parallel architectures. The importance of better understanding the relationship between algorithms and architectures is only now becoming recognized [9].

A new model useful for understanding the relationship between decomposed algorithms and data flow architectures has been presented. Named ATAMM for Algorithm To Architecture Mapping Model, the model consists of Petri net marked graphs called the algorithm marked graph, the node marked graph, and the computational marked graph. After establishing that the computational marked graph is live, safe and consistent, graph time performance measures of time between input and output (TBIO), task time (TT), and time between outputs (TBO) are defined. Then lower bounds for the performance measures are calculated analytically from the modified algorithm

graph and the computational marked graph. A desighn strategy for achieving optimum time performance is proposed and illustrated with a design example.

II.2 Diagnostic Tool Development

Although the ATAMM model is not complicated in principle, the execution of a system modelled with it becomes hardly tractable when both the number of nodes as the number of resources increase. Therefore, it is necessary to have Diagnostic Tools to explore the execution of a given algorithm. One of the important parameters necessary to observe is concurrency. Concurrency is a measure of the number of resources that work at the same time for a specified length of execution of an algorithm. Other parameters include TBIO (Time Between Input and Output), TBO (Time Between Outputs), and TBI (Time Between Inputs). These parameters refere to the time performance of the system: the elapsed time between when input data is read and its corresponding output data is written (TBIO), the time elapsed between repetitive output writings (TBO), and the time elapsed between repetitive inputs data readings (TBI). Another necessary measurements are the time the system takes and the different states it goes through to reach steady state.

The Analyzer, a computer program, provides measurement of the items denoted above. The input to the program is a file containing a sequential account of the execution of a concurrent system. It displays the activity of the individual nodes of a graph. This display is drawn on a common time axis for easy reading of the concurrent execution of nodes. An alternate display is the plotting of the activity of the resources versus time. The program also displays the function of concurrency versus time which is now called Total Resource Utilization Envelope. For individual data packets, the program displays the values of TBIO, TBO and TBI. It also reports

general statistics of the transitions per node. This program is primarily to be used for post-execution detailed analysis of the execution of an algorithm.

Another computer program, the Graph Simulation/Analyzer, provides not only simulation of the execution of an algorithm but also analysis of data immediately after execution. It generates the sequential files containing firing of transitions in the CMG (Computational Marked Graph) to be analyzed by the Analyzer, the program described above. It also generates files with average values of TBO, TBI and TBIO. The simulation module has been improved so that it may include random variables as the values of the transitions in the CMG. It accepts as input an ASCII file containing a description of the topology of a graph, transition time assignments, priority assignment, initial marking, number of resources, etc.

II.3 Testbed Development

A multiprocessor operating system has been developed based on the ATAMM specifications. It is the third prototype system to have been built in the past two years. The motiviation for this is to give further credibility to ATAMM through system validation and to provide a testbed experimentation. This discussion is divided into three design phases. In the system partitioning the ATAMM model is divided into logical components. Combined, these logical components must fully represent the ATAMM description. The next phase is the hardware mapping in which the logical components are mapped into a target architecture. Necessary inter-module communications and control dialogue paths must also be specified. The multiprocessor operating system implementation is the final design phase and will be referred to briefly.

Three logical components have been isolated in the ATAMM partition; the Graph Manager (GM), Funcitonal Unit (FUN), and Global Memory (GLM). The Graph Manager is responsible for implementing the state transitions of the processes. It must monitor all token movement within the CMG required to determine the fireability of a process. When a process can fire the Graph Manager must assign the first available Functional Unit to that process. The Functional Unit will then execute all three NMG transitions for that particular process. It must also, via interrupt, update all important token movement within the NMG to the Graph Manager. As a Functional Unit can be assigned to any process, it must also have the code available for the computation of every process in the AMG. The Global Memory is the final logical component in the partition and is responsible for storing data associated with all Output Full edges in the CMG. Because of this the it must have a communications path to all Functional Units for both the reading and writing of data.

The three prototype multiprocessor operating systems previously mentioned have all had different hardware mappings. Each new mapping was guided through observaitons made in the development of the previous mapping. In the current mapping all three logical components are distributed within each hardware module. The hardware modules chosen are IBM PC/AT's and are connected on an Ethernet Local Area Network. This mapping presents two advantages over the previous two in which the logical components were not completely distributed. First, the redundancy of all logical components provides a greater degree of fault tolerance. Secondly, a reduction of inter-module communications, the major bottleneck in multiprocessor design, is expected as the logical components all reside in the same hardware module.

The final step in the design process is to develop a multiprocessor operation system to implement the logical components as designated by the hardware mapping. In addition to the hardware modules, a Sink/Source node module was designed for the system initialization and monitoring. It is also responsible for injecting input data into the system and for receiving output data. The resulting multiprocessor has been successfully developed and is currently undergoing tests for ATAMM validation. Initial results are positive and all tests should be completed by the end of August.

III.0 OPTIMUM TIME PERFORMANCE

III.1 Introduction

The development of a new graph theoretic model for describing the relation between a decomposed algorithm and its execution in a data flow environment is presented. Performance measures of computing speed and throughput capacity are defined. Lower bounds for these performance measures are established. In Subsection III.2 of this report, the modeling process to describe algorithms in data flow architectures, ATAMM, is presented. The model consists of three Petri net marked graphs called the algorithm marked graph (AMG), the node marked graph (NMG), and the computational marked graph (CMG). In Subsection III.3, the operating characteristics of these graphs are investigated. A state variable description is presented and used to establish sthe graph properties of reachability, liveness and safeness. Time performance measures for concurrent processing are defined in Subsection III.4. The ATAMM model is used as the basis for calculating analytically lower bounds for these performance measures. Then in Subsection III.5, an operating strategy which achieves optimum time performance is developed. Several exa mples are presented to illustrate these concepts.

III.2 ATAMM Model Development

In this subsection the ATAMM model to describe concurrent processing of decomposed algorithm is presented. The model consists of a set of Petri net marked graphs which incorporate general specifications of communication and processing associoated with each computational event in a data flow architecture. First, a detailed description of the problem context is stated. This is followed by the definition of the ATAMM model consisting of

the algorithm marked graph, the node marked graph, and the computational marked graph. Some familiarity with Petri nets [10] and marked graphs [11] is assumed in this presentation.

The problems of interest are decision-free, computationally complex problems as are often found in signal processing and control applications. A problem description normally results in the definition of a function given by the triple (X,Y,F). The set X represents the set of admissible inputs, the set Y represents the set of admissible outputs, and F:X->Y is the rule of correspondence which unambiguously assigns exactly one element from Y to each element of X. Associated with a computational problem is one or more algorithms. An algorithm is an explicit mathematical statement, expressed as an ordered set of primitive operations, which explains how to implement the rule of correspondence F. In general, a given problem can be decomposed by several different primitive operator sets. Also, for a given primitive operator set, there are often different orderings of primitive operations which can be specified to carry out the problem. Of special interest are algorithm decompositions in which two or more primitive operations can be performed concurrently. For such decompositions, the potential exists for decreasing the computational time required to solve the problem by increasing the computational resources which implement the primitive operations program storage and temporary input and output data containers.

The hardware environment for executing the decomposed algorithms is assumed to consist of R identical processors or functional units (FUNs) where R has a value in the range of two to twenty. This range of resources is suggested for practical reasons due to the large-grained aspect of the algorithm decomposition and the need to maintain small communication times relative to process times. Each FUN is a processor having local memory for

program storage and temporary input andoutput data containers. Each FUN can execute any algorithm primitive operation. The FUNs share a common global memory (GLM) which may be either centralized or distributed. The coordination of FUNs in relation to data and control flow is directed by the graph manager (GRM). The GRM also may be centralized or distributed. Output created by the completion of a primitive operation is placed into global memory only after the output data containers have been emptied. That is, outputs must be consumed as inputs to successor primitive operations before allowing new data to fill the output locations. Assignment of a functional unit to a specific algorithm primitive operation is made by the GRM only when all inputs required by the operation are available in global memory and a functional unit is available.

An algorithm marked graph is a marked graph which represents a specific algorithm decomposition. Vertices of the algorithm graph are in a one-to-one correspondence with each occurrence of a primitive operation. The algorithm graph contains an edge (i,j) directed from vertex i to vertex j if the output of primitive operation i is an input for primitive operation j. Edge (i,j) is marked with a token if an output from primitive operator i is available as an input to primitive operator j. When constructing an algorithm graph, vertices (primitive operations) are displayed as circles, and edges (input-output signals) are displayed as directed line segments connecting appropriate vertices. The presence of a token on an edge is indicated by a solid dot placed on the edge. Source transitions and sink transitions for input and output signals are represented as squares. Sources for constants are not usually included in the algorithm marked graph; however, triangles are used for this purpose when necessary.

To illustrate the construction of an algorithm marked graph, consider the problem of computing the output of a discrete linear system given a sequence of inputs to the system. Let the system be described by the state equation

$$x(k) = Ax(k-1) + Bu(k)$$

and output equation

$$y(k) = Cx(k).$$

where x is p-vector, us is an m-vector, and y is an r-vector. The primitive operations are defined as matrix multiplication and vector addition, and the natural algorithm decomposition resulting from the state equation description is selected. The algorithm marked graph for this decomposed algorithm is shown in Fig. 1. The initial marking indicates that initial condition data are available.

The algorithm marked graph is a useful tool for representing decomposed algorithms and for displaying data flow within an algorithm. However, the algorithm graph does not display procedures that a computing task. In addition, the issues of control, time performance, and resource management are not apparent in this graph. These important aspects of concurrent processing are included in the ATAMM model through the definition of two additional graphs. The node marked graph (NMG) is defined to model the execution of a primitive operation. The computational marked graph, obtained from the AMG and the NMG by a set of construction rules, integrates both the algorithm requirements and the computing environment requirements into a comprehensive graph model. These additional marked graphs are defined in the following.

The NMG is a Petri net representation of the performance of a primitive operation by a functional unit. Three primary activities, reading of input data from global memory, processing of input data to compute output data,

and writing of output data to global memory, are represented as transitions (vertices) in the NMG. Data and control flow paths are represented as places (edges), and the presence of signals is notated by tokens marking appropriate edges. The conditions for firing the process and write transitions of the NMG are as defined for a general Petri net, while the read transition has one additional condition for firing. In addition to having a token present on each incoming signal edge, a functional unit must be available for assignment to the primitive operation before the read node can fire. Once assigned, the funcitonal unit is used to implement the read, process, and write operations before being returned to a queue of available FUNs. The initial marking for an NMG consists of a single token in the "process ready" place. The NMG model is shown in Fig. 2.

A computational marked graph (CMG) is constructed from the AMG and the NMG by the following rules.

1. Source and sink nodes in the algorithm marked graph are represented by source and sink nodes in the CMG.
2. Nodes corresponding to primitive operations in the algorithm marked graph are represented by NMGs in the CMG.
3. Edges in the algorithm marked graph are represented by edge pairs, one forward directed for data flow and one backward directed for control flow, in the CMG. The initial marking for the edge pair consists of a single token in the forward-directed place if data are available, or a single token in the backward-directed place if data are not available.

The play of the CMG proceeds according to the following graph rules.

1. A node is enabled when all incoming edges are marked with a token. An enabled node fires by encumbering one token from each incoming

edge, delaying for some specified transition time, and then depositing one token on each outgoing edge

2. A source node and a sink node fire when enabled without regard for the availability of a FUN.
3. A primitive operation is initiated when the read node of an NMG is enabled and a FUN is available for assignment to the NMG. A FUN remains assigned to an NMG until completion of the firing of the write node of the NMG.

In order to illustrate the construction of a computational marked graph, the CMG corresponding to the algorithm marked graph of Fig. 1 is shown in Fig. 3. The computational marked graph is useful because it clearly displays the data and control flow which must occur in any hardware implementation of the model process, and because it clearly displays the data and control flow which must occur in any hardware implementation of the model process, and because it provides a hardware independent context in which to evaluate process performance.

The complete ATAMM model consists of the algorithm marked graph, the node marked graph, and the computational marked graph. A pictorial display of this model is shown in Fig. 4. In the next subsection, important operating characterists of the ATAMM model are investigated.

III.3 Model Characteristics

In the previous subsection, a marked graph model consisting of the AMG, the NMG, and the CMG is defined as a means to describe concurrent processing of decomposed algorithms. In this subsection the ATAMM model is studied analytically to determine important graph operating characteristics. First, a state description which expresses the next graph marking as a function of

the present marking and a vector indicating which transition is to be fired is developed. Then, the marked graph properties of reachability, liveness, and safeness are considered for the CMG. Two excellent papers by Murata [11], [12] on properties of marked graphs are the source for much of the material presented in the subsection.

Let G be a marked graph consisting of m places and n transitions. The m-vector M_k denotes the marking vector for G resulting from the firing of some sequence of k transitions. The following two definitions are necessary to develop the state description of the CMG.

Definition 1: Complete Incidence Matrix. The complete incidence matrix for a marked graph G is the (nxm) matrix $A = [a_{ij}]$ having rows corresponding to transitions, columns corresponding to places, and where

$$a_{ij} = \begin{cases} +1(-1) & \text{if place j is incident at transition i and directed out of (into) the transition} \\ & \text{if place j is not incident at transition j} \end{cases}$$

Definition 2: Elementary Firing Vector. An elementary firing vector u_k is an n-vector having all zero entries except for the ith component which is 1 denoting that transition i is the kth transition to fire in some transition firing sequence.

To gain insight to the state equation description, it is helpful to consider the firing of transition k. If $a_{ki} = -1(+1)$, place i is an input (output) place to transition k. Therefore, transition k is enabled if M(i) = 1 for each input place. When transition k fires, one token is removed from each input place and one token is added to each output place. These observations lead to the following next state description for a marked graph.

Property 1: Next State Description. For a marked graph G with present marking vector M_{k-1} and elementary firing vector u_k, the next marking vector is given by

$$M_k = M_{k-1} + A^T u_k.$$

The next state description can be used to express the graph marking resulting from the application of sequences of elementary firing vectors. This is done in the next definition and property.

Definition 3: Firing Count Vector. Let $(u_1, u_2, \ldots, u_d)$ be a sequence of elementary firing vectors taking a marked graph G from an initial marking M_0 to a destination marking M_d. The firing count vector x_d for this firing sequence is defined by

$$x_d = \sum_{k=1}^{d} u_k.$$

Property 2: State Equation Description. For a marked graph G with initial marking vector M_0, the marking vector resulting from the application of elementary firing vector sequence $(u_1, u_2, \ldots, u_d)$ is given by

$$M_d = M_0 + A^T x_d.$$

Using the state description of a marked graph as a basis, the property of reachability is investigated. Necessary and sufficient conditions for a CMG marking vector to be reachable from an initial marking are established,

and it is shown that the number of tokens contained in any directed circuit of the CMG is invariant under transition firings.

Definition 4: Reachability. A marking M_d is reachable from an initial marking M_0 if there exists a sequence of elementary firing vectors that transforms M_0 to M_d.

The following definition is required to state the reachability conditions for a CMG.

Definition 5: Fundamental Circuit Matrix. Let T be a tree of a connected marked graph G. The set of (m-n+1) circuits, each uniquely formed by appending one cotree edge to the tree, is called the set of fundamental circuits of G for tree T [13]. The fundamental circuit matrix for G for tree T is the2(m-n+1 x (m) matrix $B_f = [b_{ij}]$ having rows corresponding to fundamental circuits, columns corresponding to places, and where

$$b_{ij} = \begin{cases} +1(-1) & \text{if place j is containedin f-circuit i and the place and circuit drections agree (disagree)} \\ 0 & \text{if place j is not contained in f-circuit i..} \end{cases}$$

Property 3: Reachability in the CMG. In a computational marked graph G, a marking M_d is reach;able from an initial marking M_0 if and only if $B_f M_d = B_f M_0$, where B_f is a fundamental circuit matrix for G.

Proof. It is shown in [11] (Theorem 3) that the property is true for marked graphs containing no token-free directed circuits. By the construction rules for the CMG, directed circuits occur in exactly four ways. First, each NMG consists of a directed circuit which contains an initial marking token in the "process ready" place. Second, a directed circuit is formed each time an NMG is linked to another NMG. Since one of the two linking

places contains an initial marking token and both places are contained in the circuit, this circuit is never token free. Third, directed circuits exist in the CMG corresponding to interconnected feedforward paths in the algorithm marked graph. Each such circuit contains one or more backward-directed control edge containing one initial marking token. Fourth, directed circuits exist in the CMG corresponding to directed circuits in the algorithm marked graph. Each such circuit contains exactly one forward-directed edge containing one initial marking token representing initial condition data. Therefore, the CMG contains no token-free directed circuits and the property follows.

As a direct consequence of the reachability property of the CMG, it can be shown that the number of tokens in any directed circuit is constant. This characteristic is stated as Property 4.

Property 4: Token Count Invariance. In a CMG, the number of tokens contained in a directed circuit is invariant under transition firing.

Proof. Consider a directed circuit C of a CMG. The entries in the row of a circuit matrix B corresponding to C are ± 1 in columns representing edges in C and are 0 otherwise. If M is a marking vector, the component of BM corresponding to C is equal to the number of tokens in directed circuit C under marking M. Therefore, if M_d is any marking reachable from an initial marking M_0, it follows from Property 3 that $BM_d = BM_0$. That is, the number tokens in directed circuit C under initial marking M_0 is equal to the number of tokens under any marking M_d reachable from M_0. This completes the proof.

Next, liveness and a closely related property called consistency are considered. It is shown that the CMG is live and consistent.

Definition 6: Liveness. A marked graph G is said to be live for a marking M if, for all markings reachable from M, it is possible to fire any transition of G by progressing through some transition firing sequence.

Property 5: Liveness in the CMG. The computational marked graph is live for all appropriate initial markling vectors.

Proof. It is shown in [12] (Property 2) that a marked graph G is live for a marking M if and only if G contains no token-free directed circuits in marking M. As stated in the proof of Property 3, for all appropriate initial markings M_0, the CMG contains no token-free directed circuits. Therefore, the property follows.

Definition 7: Consistency. A marked graph G is said to be consistent if there exists a marking M and a transition firing sequence S from M back to M such that every transition occurs at least once is S.

Property 6: Consistency in CMG. A connected computational marked graph G is consistent. In addition, each transition of G occurs an equal number of times in a firing sequence from a marking M back to M.

Proof. From Property 2, if a CMG is cosistent, then there exists a marking $M_d = M_0$ and a firing count vector $x_d > 0$ such that $A^T x_d = 0$. The converse is also true. The incidence matrix for a marked graph G is an (n x m) matrix A. If G is connected, then it is known [13] that the rank of A is n-1, and thus the null space of A^T has dimension one. It is observed that each row of A^T has one (1), one (-1), and all remaining terms are (0). Therefore, if C_j denotes the j^{th} column of A^T, it follows that

$$\sum_{j=1}^{n} C_j = 0.$$

Thus, there exists a vector $x_d = [k\ k\ \ldots\ k]^T$, $k > 0$, which uniquely satisfies $A^T x_d = 0$. This completes the proof.

The final graph property considered in this section is safeness. This property is first defined, and then it is shown that CMG is safe.

Definition 8: Safeness. A marked graph G is said to be safe for marking M if, for all markings reachable from M, no place contains more than one token.

Property 7: Safeness in the CMG. The computational marked graph is safe for all appropriate initial marking vectors.

Proof. By Property 4, the token count for each directed circuit of the CMG is invariant under transition firing. Therefore it is sufficient to show that each edge of the CMG belongs to at least one directed circuit containing a single token. By the construction rules for the CMG, all CMG edges can be classified into two groups, NMG edges and linking edges. NMG edges occur in groups of three and always form a directed circuit containing one token. Linking edges occur in pairs, one forward directed and one backward directed, and also form a directed circuit with the forward directed edges of the NMG. One of the linking edges, but not both, always contains one token while the forward directed edges of the NMG contain no tokens. Therefore, each edge of the CMG is contained in a directed circuit with one token, and the property follows.

III.4 Performance Analysis

The importance of the ATAMM model is that it establishes a context in which to investigate the performance of decomposed algorithms in multiprocessor data flow architectures. In this subsection, performance measures indicating computing speed and throughput capacity are defined. Bounds for

these quantities are calculated analytically from the algorithm marked graph and the computational marked graph. This information is essential for efficiently matching algorithm decompositions with architecture implementations. The work presented in this subsection is an interesting application and extension of recent investigations of the performance of Petri Nets [14], [15] and marked graphs [16].

It is assumed that a decomposed algorithm is implemented in a multiprocessor architecture containing R computing resources or functional units. Each functional unit is capable of performing any of the primitive operations whose sequence defines the decomposition. A computational task consists of completing the algorithm for one frame of data and is initiated when an input data token from the source node is encumbered. Task output occurs when a corresponding output data token is deposited at the output sink node. A task is completed when all computing associoated with the task is completed. It should be noted that task output and task completion do not always coincide. In many iterative signal processing algorithms, computing to generate initial conditions for the next iteration often occurs after an output has been calculated. Task completion is usually indicated in the AMG or CMG by the return of the graph to some steady-state initial marking. To facilitate measurement of throughput capacity, it is assumed that tasks are repeated periodically with new input data sets. New data sets are available continuously as input tokens from the input source node. Included in this problem class are iterative algorithms where the present task requires as inputs data from previous task calculations.

Concurrency in this problem setting occurs in two ways. First, different functional units may perform simultaneously several primitive operations belonging to a single task. This type of concurrency is referred to as

vertical concurrency. Vertical concurrency has a direct effect on task computing speed. It is limited by the number of primitive operations that can be performed simultaneously in a given algorithm decomposition, and by the number of functional units available to perform the primitive operations. Second, different functional units may perform simultaneously r primitive operations belonging to different tasks sequentially input to the computing system. Called horizontal concurrency, this type of concurrency has a direct effect on throughput capacity. It is limited by the capacity of the graph to accommodate additional task inputs, and by the number of functional units available to implement the tasks. In the following it is shown that the process of algorithm decomposition imposes bounds on the amount of vertical concurrecy and horizontal concurrency possible in a given problem. If sufficient computing resources are available, operation at these bounds can be achieved. If the number of computing resources is limited, the bounds cannot be reached simultaneously and trade-offs between the amount of vertical concurrecy and horizontal concurrency are possible.

Three performance measures for concurrent processing are defined. The first two parameters, TBIO and TT, are indicators of computing speed and reflect the degree of vertical concurrency. The third parameter, TBO, is a measure of throughput capacity and thus reflects the degree of horizontal and vertical concurrency.

Definition 9: TBIO. The performance measure TBIO is the computing time which elapses between a task input and the corresponding task output.

Definition 10: TT. The performance measure TT is the computing time which elapses between a task input and the completion of all computation associated with that task.

Definition 11: TBO. The performance measure TBO is the computing time which elapses between successive task outputs when the graph is operating periodically in steady-state.

The remainder of this section is devoted to developing lower bounds for these performance measures.

Let G denote an algorithm marked graph representing as decomposed algorithm. The lower bound for TBIO is the shortest time required for a data token from the data input source to propagate through the graph to the data output sink. Similiarly, the lower bound for TT is the shortest time required to complete all computing activity initiated by the injection of a data input source. These shortest times are the actual performance times when only a single task is active in the graph during any time interval (no horizontal concurrency), and as many computing resources as are required are available (maximum vertical concurrency). Under these operating conditions, lower bounds for TBIO and TT are calculated by identifying certain longest paths in a graph obtained from the algorithm marked graph. This new graph, called the modified algorithm graph G_M, is defined and then used to determine lower bounds for TBIO and TT.

Definition 12: Modified Algorithm Graph. Let p_i be a place of G, directed from transition t_r to transition t_s, which contains a token of the initial marking. The modified algorithm graph G_M is obtained from the graph G by the following construction rules.

1. Place p_i is deleted from G.
2. A new place p_{i1}, directed from the data input source to transition t_s, is added to G.
3. A new output sink s_i different from all other output sinks, and a new place p_{i2}, directed from transition t_r to s_i, are added to G.

4. The above rules are repeated for each place of G containing a token of the initial marking.

Lower bounds for TBIO and TT are presented in Theorem 1 and Theorem 2 respectively.

Theorem 1: Lower Bound for TBIO. Let P_I be the i^{th} directed path in G_M from the data input source to the data output sink, and let $T(P_i)$ denote the sum of transition times for transitions contained in P_i. Then,

$$TBIO_{LB} = Max \{T(P_i)\},$$

where the maximum is taken over all paths P_i graph G_M.

Proof. Without loss of generality, let t_f be the last transition in all paths P_i directed from the data input source to the data output sink. Transition t_f is enabled when each input place for t_f contains a token. Since by assumption a computing resource is available, t_f fires as soon as it becomes enabled. Let p_q be the last input place for t_f to acquire a token, and let t_g be the input transition for place p_q. Continuing this labeling procedure results in a backward path construction process. This process is repeated, first at t_g, and then at each succeeding transition until the data input source is reached, identifying a path p_j. By the construction process for the path, it is clear that $T(P_j) = Max \{T(P_i)\}$, where the maximum is over all paths P_i in G_M. It is also clear that $TBIO_{LB}$ can be no shorter than $T(P_j)$ so that $TBIO_{LB} \geq T(P_j)$. Since a computing resource is available when each transition in P_j is enabled, the time between input and corresponding output can be no longer than $T(P_j)$ so that $TBIO_{LB} \leq T(P_j)$. Therefore, $TBIO_{LB} = T(P_j) = Max \{T(P_i)\}$, where the maximum is over all paths P_i

in G_M. This completes the proof.

Theorem 2: Lower Bound for TT. Let P_i be the i^{th} directed path in G_M from the data input source to any output sink, and let $T(P_i)$ denote the sum of transition times of transitions contained in P_i. Then,

$$TT_{LB} = \text{Max}\{T(P_i)\}$$

where the maximum is taken over all paths P_i in graph G_M.

Proof. By the construction rules for graph G_M, a task is initiated when input data tokens are input from the data input source, and is completed when all output sinks have accepted tokens. Therefore, TT is the time which elapses from injection of input tokens to the arrival of a token at the last fired output sink. Let $T(P_t) = \text{Max}\{T(P_i)\}$, P_i in GM, be the longest path time of paths from the data input source s_I to any output sink, say s_t. Since a token must reach sink s_t before a task is completed, it follows that $TT_{LB} \geq T(P_t)$. Since a resource is available for each transition to fire when enabled, and since P_t is the longest path in G_M, it also follows that $TT_{LB} \leq T(P_t)$. Therefore, $TT_{LB} = T(P_t) = \text{Max}\{T(P_i)\}$, where the maximum is over all paths P_i in G_M. This completes the proof.

To illustrate the application of Theorem 1 and Theorem 2, $TBIO_{LB}$ and TT_{LB} are computed for the algorithm graph shown in Fig. 1. For this example, the following transition times are assumed: T(1) = 4, T(2) = 1, T(3) = 5, and T(4) = 6. The modified algorithm graph coresponding to Fig. 1 is shown in Fig. 5. The modified algorithm graph contains two paths directed from the data input source s_I to the data output sink s_O. Path P_1 consists of edge set $\{1, 2, 3, 4\}$ with $T(P_1) = 10$, and path P_2 consists of edge set

$\{5\text{-}1, 3, 4\}$ With $T(P_2) = 6$. Therefore, since $T(P_1) > T(P_2)$, path P_1 determines the lower bound for TBIO and $TBIO_{LB} = 10$. The modified algorithm graph contains two additional directed paths from the data input source s_I to the output sink s_5. Path P_3 consists of edge set $\{1, 2, 6, 5\text{-}2\}$ with $T(P_3) = 11$, and path P_4 consists of edge set $\{5\text{-}1, 6, 5\text{-}2\}$ with $T(P_4) = 7$. Since $T(P_3) > T(P_1) > T(P_4) > T(P_2)$, path P_3 determines the lower bound for TT and $TT_{LB} = 11$.

Next a lower bound for the performance measure TBO is presented. Let G be a computational marked graph representing a decomposed algorithm. It is assumed that operating conditions for G are set to maximize horizontal concurrency. That is, data tokens are continuously available at the data input source, and as many computing resources as needed can be called to perform primitive operations. With these conditions, the graph plays periodically in steady-state, and TBO_{LB} is the shortest time possible between successive outputs.

Theorem 3: Lower Bound for TBO. Let G be a computational marked graph and let C_i be the ith directed circuit in G. The notation $T(C_i)$ denotes the sum of transition times of transitions contained in C_i, and $M(C_i)$ denotes the number of tokens contained in C_i. Then,

$$TBO_{LB} = \text{Max} \{T(C_i)/M(C_i)\},$$

where the maximum is taken over all directed circuits in G.

Proof. Without loss of generality, let t_f be the output transition in G so that an output is produced each time t_f completes the firing. Then TBO_{LB} is the minimum firing period of transition t_f. By Property 6, G is consistent so that all transitions of G fire periodically with minimum period TBO_{LB}.

It is shown in [12] (pp. 58-60) that the minimum firing period of each transition of a marked graph is given by $\text{Max}\{T(C_i)/M(C_i)\}$, where the maximum is taken over all directed circuits C_i in G. Therefore, the theorem follows.

The computational marked graph shown in Fig. 3 is used to illustrate Theorem 3. This CMG contains many directed circuits. However, the directed circuit which contains all NMG nodes of transitions 2 and 4 contains only one token and maximizes the ratio $T(C_i)/M(C_i)$. Therefore, the shortest time possible between successive outputs in this graph is $TBO_{LB} = 7$. In the next subsection, a strategy for achieving optimum time performance is investigated.

III.5 Strategy for Optimum Time Performance

A model describing decomposed algorithms for implementation in a distributed data flow architecture is described in Subsections III.2 and III.3, and performance measures are defined in Subsection III.4. An important problem remaining is to develop an operating strategy for the ATAMM model which achieves optimum time performance with a minimum number of computing resources. Unfortunately, this problem is equivalent to a class of scheduling problems which is known to be NP-complete. Thus, there exists no algorithm for obtaining an optimum solution which is better than enumerating all possible solutions and then choosing the best one. However, an important suboptimal operating strategy which achieves optimum time performance, but possibly requires more than the minimum number of computing resources, has been developed. This strategy is presented and illustrated by example in this subsection.

When presented with continuously available input data sets, the natural behavior of a data flow architecture results in operation where new data sets are accepted as rapidly as the available resources permit. That is,

the architecture naturally operates at high levels of horizontal concurrency with the possible loss of capability for achieving high levels of vertical concurrency. This results in performance characterized by high throughput rates, $TBO=TBO_{LB}$, but relatively poor task computing speed so that $TBIO > > TBIO_{LB}$ and $TT > TT_{LB}$. In many signal processing and control applications, it is important to achieve both high throughput rate and high task computing speeds. Often, designers are willing to provide extra hardware to realize optimum time performance. The suboptimal operating strategy presented in this section results in performance having the following characteristics.

1. When $R \geq R_{Max}$, operation achieves $TBIO_{LB}$, and TBO_{LB}. R_{Max} is computed in implementing the strategy, and represents the minimum number of resources which insures maximum horizontal concurrency and maximum vertical concurrency under this strategy.
2. When $R_{Max} > R \geq R_{Min}$, operation achieves $TBIO_{LB}$ and TT_{LB}, but $TBO \geq TBO_{LB}$. The strategy preserves task computing speed or vertical concurrency at the expense of throughput rate or horizontal concurrency. R_{Min} is also computed in implementing the strategy, and represents the minimum number of resources needed to maintain vertical concurrency with limited horizontal concurrency.
3. When $R_{Min} > R \geq 1$, operation continues but performance degrades so that $TBIO \geq TBIO_{LB}$, $TT \geq TT_{LB}$, and $TBO \geq TBO_{LB}$.

Implementation of the operating strategy is illustrated in Fig. 6. All that is required is to limit the rate at which new input data are presented to the CMG. This is accomplished by adding a control transition connected in a directed circuit with the data input source. The control transition imposes a minimum delay of D time units between inputs. Delay D is chosen according to the following rule:

$$D = \begin{cases} TBO_{LB} & R \geq R_{Max} \\ TBO_{Min} & R_{Max} > R \geq R_{Min} \\ TCE & R_{Min} < R \geq 1. \end{cases}$$

TCE denotes the total computing effort required to complete the task, and TBO_{Min}, R_{Max}, and R_{Min} are computed as part of the strategy design procedure.

The operating strategy design process consists of five steps. These steps are presented and explained in the remainder of this subsection. An operating strategy is developed for the example algorithm graph shown in Fig. 7 to illustrate each step as it is presented.

Step 1. Choose a convenient transition firing rule. A rule to determine when an enabled transition in the CMG fires must be specified. A natural rule is to specify that enabled transitions fire when a computing resource is available. If conflict exists, such as when there are more enabled transitions than computing resources, then firing occurs according to a priority ordering of the transitions. For the example algorithm graph, the highest to lowest priority ordering of the transitions is chosen as (5,4,3,-7,2,6,1).

Step 2. Determine TBO_{LB}. The performance bound TBO_{LB} is found from the computational marked graph by application of Theorem 3. The CMG corresponding to the example algorithm graph is shown in Fig. 8. The directed circuit identified in this figure contains 6 transition time units and 2 tokens, and maximizes the ratio $T(C_i)/M(C_i)$ for all directed circuits. Therefore, $TBO_{LB} = 3$.

Step 3. Determine the resource utilization envelope of a single task required for maximum vertical concurrency at steady-state with TBO = TBO_{LB}.

The purpose of this step is to determine the number of computing resources required as a function of time to achieve maximum vertical concurrency in a single task. The envelope is determined by playing the graph assuming unlimited resources and an input rate of TBO_{LB} until steady-state operation is reached. The resource utilization envelope is obtained by counting the number of computing resources used for a single task during each time interval. The play of the example algorithm graph under these conditions is shown in Fig. 9, and the resulting resource utilization envelope is shown in Fig. 10.

Step 4. Stabilize the resource utilization envelope by adding control places as necessary. If the time between inputs to the CMG is increased above TBO_{LB}, the resource utilization envelope may change from that observed in Step 3. Since knowledge of the envelope is required to calculate the number of required resources, additional places are appended to the AMG and the CMG to freeze the shape of the envelope. For example, the play of the example algorithm graph of Fig. 8 with an injection time of 4 is shown in Fig. 11. At this slower injection rate, transition 6 fires one time unit earlier. To prevent time movement of transition 6, a control place directed from transition 2 to transition 6 is added. This place prevents the firing of transition 6 until transition 2 has completed firing. Thus the resource utilization envelope computed for an input period of TBO_{LB} is the envelope for all input periods $TBO > TBO_{LB}$.

Step 5. Compute R_{Max}, R_{Min}, and $TBO_{Min}(R)$ using the resource utilization envelope. R_{Max} is determined by overlaying resource utilization requirements, each delayed by TBO_{LB} with respect to the previous one, as shown in Fig. 12 for the example. R_{Max} is equal to the largest resource requirement

during any time interval within the steady state operating period. R_{Min} is the minimum number of resources necessary to insure maximum vertical concurrency with no horizontal concurrency. This number is equal to the maximum resource requirement indicated in the resource utilization envelope for a single task. For the example problem, $R_{Max} = 5$ and $R_{Min} = 3$. The value of TBO_{Min} for each resource number R between R_{Max} and R_{Min} inclusive, is determined by increasing the delay between overlapping resource utilization envelopes until the maximum resource requirement is R. TBO_{Min} is the smallest input delay to produce this resource requirement. For the example, the calculations of TBO_{Min} for R = 4 and R = 3 are illustrated in Fig. 13 and Fig. 14 respectively. The results of these calculations are $TBO_{Min}(4) = 3.5$ and $TBO_{Min}(3) = 4$.

The performance of the example algorithm graph is summarized in Fig. 15. Optimum time performance of $TBIO_{LB} = TT_{LB} = 7$ and $TBO_{LB} = 3$ is achieved for $R \geq R_{Max} = 5$. At R = 4, TBIO and TT remain at the optimum values and TBO_{Min} decreases to 3.5. At R = 3, TBIO and TT again remain at the optimum values and TBO_{Min} decreases to 4. For values of R below R_{Min}, time performance generally degrades. However, in this example TBIO and TT remain at 7 for R = 2, while TBO_{Min} decreases to 6. Finally, at R = 1, performance degrades to TBIO = TT = TBO = TCE = 10. Another perspective of algorithm performance is shown in Fig. 16. This figure displays throughput rate, (1/TBO), as a function of the number of functional units R. The peak height of each bar indicates the maximum throughput rate which can be achieved with the indicated number of processors. The bars also indicate more clearly that operation at any throughput rate less than maximum is possible for a given number of functional units. This design procedure is easily applied

to much larger algorithm graphs more representative of actual signal processing and control problems.

IV.0 DIAGNOSTIC TOOL DEVELOPMENT

IV.1 Analyzer Development

IV.1.1 Introduction

Concurrent processing is the capability of a computer system to execute two or more tasks at the same time. For example, a processor may execute a given computation at the same time that an I/O coprocessor performs an I/O operation. There are new computer architectures that organize processors in a parallel fashion requiring customized algorithms to take advantage of the parallelism of the systems. However, the models developed to describe these architectures do not adequately model the issues of scheduling, coordination, and communication (Ref. 17). On the other hand, the strategy proposed by Stoughton and Mielke (Ref. 17-19) addresses these particular issues. The strategy uses timed Petri nets (Ref. 20) to model processor behavior for each computational node of an algorithm graph.

Detailed data are needed to evaluate and study the performance of a concurrent processing system. Data such as the function of concurrency with respect to time can be investigated. Therefore, a sophisticated evaluation of the concurrent system can be performed. To achieve this objective, it is indispensable that data, such as when the processing of a data packet is initiated and when it is terminated, be available. Performance measures such as TBIO or TBO can be obtained from global information such as when an input is read by the graph and when its corresponding output is written. This kind of information can be obtained from an outside observer which monitors the system. The best information the system is able to provide is the firing of every transition of every node during execution. With these data, a more comprehensive study of a concurrent processing system can be done. Although the system itself is used to provide the information, it does not affect the

performance of the system due to the relatively low speed communication channels used in the prototype. Another method to probe the system should be devised if high speed communication channels are to be used. This chapter describes an analyzer system that yields the required evaluation. In Subsection IV.1.1, a prototype system and its communication events will be closely examined. What the Diagnostic Routines do in the Graph Manager and their effect in the overall performance is contained in Subsection IV.1.3. How information of internal events is recorded is presented in Subsection IV.1.4. In Subsection IV.1.5, generalities of the Analyzer program are examined, including what information is input to it and what is obtained as output data. In Subsections IV.1.6, IV.1.7 and IV.1.8, how the Analyzer program processes this output data to generate measurement information is presented. These measurements include TBIO (Time between Input and Output), TBI (Time between Inputs), TBO (Time between Outputs), concurrency of the computing system and general average process times. In the last two Subsections IV.1.9 and IV.1.10, a different tool is presented. This tool integrates the simulation of the system and the analyzer in one program.

IV.1.2 Prototype and its Communication Events

A prototype of a concurrent processing system was developed. It was used to prove some of the theories of the graph representation of such systems and to establish a basis for comparison of the simulation program to its hardware counterpart. The block diagram of the prototype, which was originally presented in (Ref. 17), is shown in Fig. 17.

The system consists of several S-100 units using Intel 8088 microprocessors. Each unit has I/O boards to communicate with the external world as well as 32k of random access memory (RAM). For test purposes, there are three

units acting as processing elements or Functional Units, one as the Graph Manager and one that serves as Global Memory. The communication between them is made through serial ports (Standard RS-232). An IBM Personal Computer XT (IBM-PC/XT) is used to communicate with the Graph Manager. A communication channel can be set through the Graph Manager to the Functional Units and the Global Memory.

The Graph Manager is designed to monitor the graph execution and is itself controlled by the data flow in the system. The Graph Manager keeps a record of the places in the graph as well as which functional unit is performing which process node. The Graph Manager "schedules" the assignment of a functional unit to a process node according to the priority of the nodes, functional units available and the process nodes that can be fired.

A serial communication link is set between the Graph Manager and every Functional Unit. A link is also set between the Global Memory and every Functional Unit. Serial communication between the IBM-PC/XT and the Graph Manager is used for initialization, and for controlling and monitoring of the system.

When a node which is found that can be fired, i.e., its input places are full and its output places are empty (the last requisite for single node model only), such node is assigned to a Functional Unit; i.e., that node is fired. To fire a node, a communications protocol is initiated between the Graph Manager and an available Functional Unit, as shown in Figure 18. This protocol begins with the code word D for Do; it is followed by a Task Number, the Inputs places or labels, and the Output places. This communication event is called Assign Task. This information, which is given to the Functional Unit, is taken from the graph data that are in the Graph Manager's memory. In this step a task or a node is said to be assigned to a Functional Unit.

The next piece of communication done between the assigned Functional Unit and the Graph Manager is the acknowledgement from the Functional Unit that the input places have been read (Acknowledge Input). The Functional Unit reads the data from the Global Memory using another protocol before Acknowledge Input is sent to the Graph Manager.

Process of data is started as soon as the input data are acknowledged by the Functional Unit. The unit communicates with the Graph Manager indicating that the process is finished when the process is done and that it is ready to place the output data in the Global Memory. The token information of the output places of the associated node is examined and it is verified that the output places are empty (the latter event is true only for the triple node model). The code for Outputs Empty is sent to the Functional Unit that is working on that node.

The data is written to the output places once the Functional Unit has clearance for writing. The Graph Manager is informed when the output is written and the Functional Unit is freed; i.e., the Functional Unit is in a wait state until the next task is assigned to it.

IV.1.3 Graph Manager Diagnostic Routines

The entire concurrent processing system is accessible to the Graph Manager; therefore, the Graph Manager is the most suitable subsystem to inform the outside world of what events are taking place in the concurrent system.

In order to keep a proper time record of the different events in the graph, an internal real-time clock is started simultaneously with the execution of the graph. As each event is recorded, the clock is read to register the time at which the event is taking place.

There are five different events that are recorded. These correspond to the communication events previously mentioned:

1 (F) Firing of a process node and binding to a Functional Unit. "Assign Task".

2 (I) Input places read by the Functional Unit (process node). "Acknowledge Input".

3 (P) Process done by the Functional Unit (process node).

4 (S) Output places empty. "Enable Outputs".

5 (O) Output places written by the Functional Unit (process node). "Acknowledge Output".

It should be noted that after a node and a Functional Unit have been assigned to each other, they cannot be distinguished from each other. They become one entity and is the only time when either one, the node or the Functional Unit, is considered active.

Every event is recorded in the following format:

T(clock)N(node number)(event)[functional unit number]

where (event) can be any of the next letters:

F (The node fires),

I (The input data is read),

P (The process is done),

S (The output places are empty), and

O (The output data is written).

The parameter of [functional unit number] is only written when {event} is equal to 'F.' To simplify the reading of the file, commas are inserted between every letter and number. The output file of the Simulation program (Ref. 21) does not require such an adjustment or addition since it is already provided with commas.

Any probe that is installed in a system for testing purposes introduces some error in the reading. The probe used here is no exception to the rule and, in order to minimize the error, a special interrupt driven routine was written. The diagnostic routines use a buffer to write the information of every graph event. This buffer is accessed every time the real-time clock is incremented and if the serial port to the IBM-PC is ready to send a character, this routine sends the next character in the buffer. If there are no characters in the buffer or the serial port is not ready the routine just increments the internal clock and exits without further action. To minimize the time that would take to write the commas to the output, a post-processor program was written that inserts the commas in their proper places. Due to the low speed communication channels, this scheme is good enough to minimize any delay introduced in the system by these Diagnostic Subroutines.

IV.1.4 Sequential Account for Concurrent Processing

All the events that are reported in the format explained in Subsection IV.3.3 are captured in a file that becomes what has been called the "ticker tape". This file contains all the necessary information to analyze the performance of the system. This file is called the FIPSO file because it accounts for Firing, Input, Process, OutStat and Output of every node in the graph. OutStat is the "enable outputs" signal sent by the Graph Manager to

the Functional Unit as shown in Fig. 18. A sample of a FIPSO file is presented in Fig. 19.

If the time between two different events is desired, the difference between the first and the last has to be computed. Or if the number of computers that were working at the same time during a certain interval is requested, the computations or procedures to obtain this number are much more complex, but not impossible to obtain.

With this kind of information, the encumbering and depositing of tokens can be monitored, although there is no direct information about these particular events. Knowing the graph topology, the depositing of tokens is done when a node writes data to its output places. The tokens are encumbered when a specific node is fired. Although it is not obvious, any type of event can be registered with this information. Getting the information can be a complex job but with the help of a specialized program this can be done rather easily.

IV.1.5 Analyzer Program

The Analyzer is a program that reads FIPSO files and obtains different data from the execution of the given graph (see Fig. 20). The data is processed to obtain such information as TBO and TBIO.

The file is read and the information is placed in a two-dimensional array, which for convenience is also called the FIPSO array. This array has fields defined as follows:

	Clock	Node 1	Node 2	Node 3 . . .
Event #1	[ ]	[ ]	[ ]	[ ] . . .
Event #2	[ ]	[ ]	[ ]	[ ] . . .
Event #3	[ ]	[ ]	[ ]	[ ] . . .
:	:	:	:	:
:	:	:	:	:
:	:	:	:	:

The clock field contains the value of the clock at the time of the event. The node field contains a code that indicates the event the node is in and, if in any, what functional unit is working on it.

The primary display of this program shows the activity of every node in the graph (see Fig. 21). The display is actually several plots aligned in time, i.e., all of them sharing the same time axis. In this way the activity of every node can be compared with the rest. For example, it can be determined if several nodes were active at the same time. Another display shows the activity of every functional unit instead of the nodes (see Fig. 22). Among other data, the concurrency of the system can be extracted at any interval in time or for the entire graph execution. In this manner, there is a display of the concurrency as a function of time. Other data are obtained and are explained in detail in the following sections.

IV.1.6 Measurement of TBIO, TBO, TBI

To measure TBIO, TBO, and TBI of the system, there is the need to know which are the input and output nodes of the system. Since this cannot be reliably extracted from the obtained information, these are parameters that have to be supplied beforehand to calculate the desired data. After the

program determines which nodes are the input and the output of the system, it proceeds to search in the matrix for occurrence of

1) When input data is read by the input node, and
2) When output data is written by the output node.

These times are recorded in another matrix for further use. Every time an output is written by the output node the time from its corresponding input is calculated and stored in the same array.

After every output has been recorded, TBI and TBO are calculated. For TBI, this is done starting from the last input entry and going down to the second input entry, substituting the ith entry by the difference of the ith entry and the i-1st entry. Calculation of TBO is done similarly, except that the output data is used instead of the input data. This output difference calculation may be expressed by

$$tOi = tOi - tOi\text{-}1 \text{ for } i = n, n\text{-}1, n\text{-}2, \ldots 2$$

where n is the number of outputs. The input difference calculation is similarly performed by

$$tIi = tIi - tIi\text{-}1 \text{ for } i = n, n\text{-}1, n\text{-}2, \ldots 2$$

where n is the number of inputs.

The display yields such information as when the system reached steady state (see Figure 23). When TBI, TBO, and TBIO do not change from one data packet to the next the system is said to be in steady state.

IV.1.7 Concurrency Measurement

Concurrency is the property associated with the capability of a computing system of executing two or more tasks at the same time. The concurrency function or what has lately been called the "Resource Utilization Envelope" can be measured or displayed in a rather simple fashion.

To obtain the concurrency information, the FIPSO array is swept in its two dimensions. The array is swept along the "event" rows and along the clock and nodes columns (see Subsection 3.4). At every row in the array, every node is checked for activity and the sum of all active nodes is obtained for that time or row. This is done for every row in the array and the function of number of resources vs. time is plotted on the screen. This is the Concurrency Display (see Fig. 24).

There is a value that is also obtained. It is called Computing Power (CP). This value is equal to the area under the curve of the Concurrency Display or the "Resource Utilization Envelope". The units of this figure is "computer-seconds". The "Resource Utilization" can be obtained by

$$RU = \frac{CP * 100}{n * T_E}$$

where RU is Resource Utilization (%), CP is Computing Power (computer-seconds), n is the number of resources (computers) and TE is Execution Time (seconds). These two quantities can be obtained for the entire execution or for a portion of it. The interval over which the evaluation is made is defined by the user.

A table showing percentages of numbers of resources concurrently used with respect to the execution time is displayed. Thus the maximum possible concurrency and its percentage with respect to the execution time can be determined.

IV.1.8 General Statistics

The different transition times have an exact value in the original simulation program (Ref. 21). However, in a hardware implementation there are some variations in these transition times. For example, a memory reading may take a longer or shorter time than expected.

There is a menu option that allows the user to get the average transition time for any node. The only parameter supplied is the node number. The program will scan the FIPSO array and calculate the average time to read the input data, process the data, wait for output place clearance and write the output data for the node indicated.

In a hardware implementation of this concurrent system, the different computers that serve as resources or functional units may have different main clocks, or can be totally different computers and of course have some differences in the time that they would take to either read, process or write data. This provides a way to obtain average time values of the activities in the system for any given node.

IV.1.9 Graph Simulation/Analyzer

The Analyzer program is an invaluable tool for the analysis of the FIPSO file of a single simulation. If the need for exploring the effect of parameter variation arises additional program support is needed. This program is

called Graph Simulation/Analyzer program. This program controls the simulation and, immediately after execution, analyzes its data to obtain the desired result or reading. Sometimes only a certain number of values are required to be analyzed and then this specialized program is ideal for automated or batch simulation and execution analysis. An overview of its features is presented in this subsection.

The Graph Simulation/Analyzer program contains basically the same simulation kernel that the original Simulation program (Ref. 21). It has been modified to provide the use of random variables as transition times.

The original Simulation program is not only a simulator but also a graph creator, i.e., the graph need not be defined when the program is called, but can be defined by the use of graphics commands. The Graph Simulation/Analyzer needs to be supplied with a graph description and simulation control (GDSC) file (see Fig. 25). This is a text file that can be created with any pure ASCII word processor and the command syntax can be found in the manual of the program in the appendix of this thesis.

The main purpose of this new program is to "schedule" a series of simulations of a graph, change parameters, and collect specific output data such as ATBIO (Average TBIO in steady state) or the usual FIPSO files. One of the advantages over the former simulation program is that most of the program setup can be in the GDSC file or, in short, the graph file. In this way, setting up a simulation can be as quick as loading the graph file and typing a few keystrokes. One of the disadvantages is that the execution of the graph cannot be seen graphically. The only parameters that can be observed are the clock and the number of outputs from the graph. Even the clock can be suppressed from updating to reduce screen update overload. A notable difference with respect to the former simulation is the capability of adding random

variables to the different transition times in a graph. The range of variation is specified by the user in the graph file.

The new program is suitable to integrate a design tool for the concurrent processing systems under study. The automatic control of the simulation routine makes the program ideal to find, through iterations, some optimum performance parameters for a given graph.

The program provides on-line context-sensitive help. At every stage where user intervention is expected, the key F1 can be typed and a window appears providing specific explanation of what the user may do at this part of the program. The help window information can be as simple as the statement of the purpose of the menu option or examples to illustrate the possible choices.

This program is expected to be changed in the future and to undergo a series of enhancements. This is the reason it was written in C language, a flexible and simple, yet powerful and easy-to-maintain language. The program can be easily expanded or modified to meet the future demands of the ongoing research.

IV.1.10 Output of the Graph Simulation/Analyzer

The Graph Simulation/Analyzer program generates only four kinds of files. These are Average Time Between Input and Output (ATBIO), Average Time Between Inputs (ATBI), Average Time Between Outputs (ATBO) and the FIPSO files. The "average" files collect data that is calculated once the system has reached the steady state. The computation of the steady state values is done by the use of a running average, in the following manner:

1- An average is computed for the first six outputs (TBIO,TBI or TBO) and stored in an average array.

2- The first of those outputs is then discarded and the seventh output is taken to form the next six outputs.

3- Another average is computed for the new six outputs and is stored in the next location of the average array.

4- This procedure is applied until there are no more outputs left to work with.

5- The next step is to find which of the computed averages is within a +/- 1% of its predecessor.

6- An overall average is calculated beginning with this predecessor up to the last average and this is the ATBIO, ATBI or ATBO.

The FIPSO files are obtained the usual way, that is, from the recording of every event, every event code is translated to text and the FIPSO file is created. This file contents can be examined in the Analyzer as explained in the last sections.

There are some instances when, although the steady state has been reached, the program will print "N/SS" (Non-Steady State) instead of the value sought. This usually occurs because the running average has too few outputs to work with and the reaching of steady state is hidden in one of the averages, i.e., the +/- 1% is too restrictive to detect it. Another error message that can be given is :"N/EO," meaning "Not Enough Outputs." The reason for this message is that there are less than nine outputs to work with and it makes it difficult to calculate the average.

The method of running averages is adequate to find when the graph reaches steady state. However, it requires many graph outputs which may create a great time burden in terms of simulation time. These computation factors

depend on the number of nodes of the graph, execution time and number of resources available.

V.0 EXPERIMENTAL RESULTS

V.1 Introduction

This Subsection presents the use of the Analyzer and the Graph Simulation/Analyzer programs to evaluate the performance of two different graphs. In Subsection IV.4.1, a graph with parallel paths is investigated. TBOLB and TBIOLB are calculated and a simulation of the system is performed. Analysis of the output data is used to obtain the minimum number of resources necessary to obtain maximum performance regardless of priority assignment. Subsection IV.4.2 is dedicated to investigate a graph with iterative loops. The same data are obtained as in Subsection IV.4.1. Subsection IV.4.3 presents two performance factors based upon TBOLB and TBIOLB.

V.2 Graphs With Parallel Paths

Graphs with parallel paths are important due to the possibility of high concurrency in the execution of tasks. Fig. 26 presents an example of a graph with three parallel paths. This example is used to illustrate the calculation of TBO_{LB} and $TBIO_{LB}$.

The first step to calculate TBOLB and TBIOLB is to choose a Node Marked Graph. The Single-Node model is selected because the resulting CMG is deadlock free. The second step is to obtain the CMG for the given graph. This is shown in Fig. 27. The third step is to obtain the circuit that takes the longest time to execute in order to obtain and get TBO_{LB}. Fig. 28 shows the circuit with the longest time per token. TBOLB is equal to 1065 time units. As the fourth step, the path from the input to the output of the graph with

the longest time has to be located. This is shown in Fig. 29. This time is $TBIO_{LB}$ and is equal to 2240 time units. The fifth step is to calculate the data injection rate which is controlled by the input source node. The time that has to be associated with this node is equal to the inverse of the input injection rate. To obtain the effective input rate to the graph, it is necessary to consider the input read time of the input node. The source node will fire when a token is placed at its control edge. This is done when the input read time of the input node is over. Therefore, the source node write time is equal to

$$\text{Write time} = TBO_{LB} - \text{Input read time (Input Node)}.$$

The effective input rate to the graph is

$$IR = 1/(TBO_{LB} - t_{IN1})$$

where IR is the input rate, and tIN1 is the input read time of node 1. Since TBO_{LB} is 1065 and I_{N1} is 140, the source node write time is 925.

V.2.1 Simulation

The simulation is performed with the calculated data for all possible number of resources. The simulation is executed for one resource, two resources and so on, up to seven resources. The data is input to the Graph Simulation/Analyzer by means of a Graph Description and Simulation Control file. The simulation is stopped when the graph has processed fifteen data packets. The GDSC file used to simulate the example is presented in Fig. 30.

Average TBO, Average TBIO and the FIPSO files are gathered for every simulation cycle. Resource Utilization (RU) and maximum number of resources concurrently used are obtained from these files.

The simulation was also run for another priority assignment. The former priority assignment tries to output as many data packets as possible; the latter tries to load the graph to its maximum before an output is written. The first priority assignment has its highest priorities toward the output of the graph, i.e., the closer to the output the higher the priority. In this way, the highest priority in the graph is to process and output data. The system tries to output data as soon as possible. The second priority assignment tries to input as much data as it can before data is output. The closer a node is to the input of the graph the higher is its priority.

V.2.2 Analysis of Output Data

The Graph Simulation/Analyzer and Analyzer data are tabulated in Tables 1 and 2. The computing power is about the same for every case since it is the total computing power required for processing fifteen data packets. The resource utilization decreases with the increase of number of resources. The resource utilization is almost the same for one and two resources. For three and more resources the resource utilization decreases more drastically for a change of one resource. For every resource added to the system the resource utilization is reduced by about ten percent.

TBOLB is closely achieved using more than four resources. The small difference is due to the overhead time introduced by the Graph Manager, or the Simulation, in the scanning and firing of the nodes of the graph. TBIOLB is obtained using more than two resources. Again, the difference with respect to the calculated value is due to the scanning of the graph.

This value of TBOLB was obtained for two different priority assignments. The value of TBOLB is not calculated based on priority assignment but on the transition times in the circuits of the graph. If it is obtained for a given number of resources, it should be maintained regardless of the priority assignment for at least the same number of resources.

The maximum number of resources used concurrently is five. After five resources there is no effect on adding resources except to lower the resource utilization. This graph can be executed at its optimum performance with five resources.

V.2.3 Minimum Number of Resources for Maximum Performance

Two important values are observed in Table 1. These are the minimum number of resources necessary to obtain TBO_{LB} and the minimum number of resources necessary to obtain $TBIO_{LB}$. TBO_{LB} is attained for at least five resources and $TBIO_{LB}$ is attained for at least three resources. The minimum number of resources for maximum performance is five since with this number of resources TBO_{LB} and $TBIO_{LB}$ is obtained. This minimum number of resources coincides with the maximum concurrency in the graph. This value has been obtained, by theoretical means, by the ODU research team and has been called Rmax.

It is important to test if this minimum number of resources is independent of priority assignment. The simulation of this graph was run for five resources and for every possible priority assignment. It turned out that the maximum performance was obtained for every priority assignment. This test method is not recommended as a common practice since it requires too many hours of simulation execution. It was done here as an exercise. It was done

to test that this minimum number of resources is independent of priority assignment for this example.

The minimum number of resources at which the TBIOLB is preserved is not priority independent. A priority can be found at which, for this number of resources, TBIO is higher than $TBIO_{LB}$. Table 3 shows the results for the same graph with a different priority assignment than the last two. The minimum number of resources at which $TBIO_{LB}$ is preserved is four instead of three as in the last two examples of priority assignments.

It should be noted that the first two simulations performed in the graph did not require more than thirty minutes, making the use of the Graph Simulation/Analyzer and the Analyzer a viable method to evaluate the performance of a given algorithm graph.

V.2.4 Graphs with Iterative Loops

Graphs with iterative loops belong to another class of graphs that is important to the ongoing research. These kinds of algorithm graphs are found primarily in applications such as digital signal processing or control systems, where data from predecessor cycles are needed for computation of a present data packet. Figure 31 presents an example of a graph with iterative loops.

The Single-Node model is also used in this example to model the nodes in the graph. Figure 32 shows the resulting CMG, using the Single-Node model, of the graph..

The circuit with the longest time per token in the CMG is located in either of the iterative loops, nodes 2 and 5, or nodes 3 and 6. Since there is only one token in the circuit, the value of TBO_{LB} is 960 time units. The effective write time of the input source is equal to TBO_{LB} less the read time

of the input node. The value of the write time of the source node is 890 time units.

Following the procedure described in Section 2.8, nodes 5 and 6 are eliminated to calculate $TBIO_{LB}$. The value of $TBIO_{LB}$ is equal to the sum of the times from the input source to the output sink. This value is 1600 time units.

V.2.5 Simulation

The simulation is performed with the calculated data for all possible numbers of resources. The simulation is executed for one resource, two resources and so on, up to six resources. The data is input to the Graph Simulation/Analyzer by means of a Graph Description and Simulation Control file. The simulation is stopped when the graph has processed fifteen data packets. The GDSC file used for this example is presented in Fig. 33.

Average TBI, Average TBO, Average TBIO and the FIPSO files are gathered for every simulation cycle. Resource Utilization (RU) and maximum number of resources used concurrently are obtained from these files.

The simulation was run for two priority assignments. This difference in priority assignments was explained in Subsection IV.4.1.1.

V.2.6 Analysis of Output Data

The Graph Simulation/Analyzer and Analyzer data are tabulated in Tables 4 and 5. R_{max} is equal to three for this graph with iterative loops. Both TBO and TBIO degrade for numbers of resources less than R_{max}. This is different from the case of the example of Subsection IV.4.1 in which only TBO degrades below R_{max} (in the mentioned example TBOLB is also attained for one and two resources below R_{max}). For the first priority assignment $TBIO_{LB}$ is still

obtained for two resources, but for the second it degrades. This behavior indicates that, for this graph, TBIO is priority dependent below R_{max}.

There is a difference of ten or eleven time units between ATBI and ATBO which is not expected since ATBI and ATBO should be equal for the conditions of the simulation. There is also an increase in the average of TBIO with respect to ATBIO for two resources in the first priority assignment. A more detailed observation of the execution in the Analyzer reveals that the difference between TBO and TBI is being added to TBIO at every data packet. Every time a data packet is injected in the graph, it takes ten more time units to arrive to the output than the precedent data packet. This is the reason of ATBIO to be much higher than expected. The reason of the difference between ATBI and ATBO can be observed in the Analyzer. The critical circuit, nodes two and five, takes more time than calculated due to the scanning of the nodes in the graph. This increase is directly applied to TBO, but TBI continues being the same that was calculated theoretically.

The source write time was incremented to 900 and the simulation was run again. The results are as expected: ATBI is 975, ATBO is 975, and ATBIO is 1620 for R_{max}.

The increase in the source write time is an experimental adjustment to obtain the best possible performance. This yields an expression for a lower bound TBO adjusted to compensate for system overhead during the execution:

$$TBO_{LBA} = TBO_{LB} + E$$

where TBO_{LBA} is the adjusted lower bound for TBO, and E is the adjustment factor obtained from the simulation of the graph, or in the case of a hardware system, the one obtained by executing the graph.

It should be noted that this adjustment factor, E, was not necessary for the example of Subsection IV.4.1. The two graphs of the examples are from two different classes of graphs. The graph of Subsection IV.4.1 belongs to a class that has its input circuit directly "coupled" to the critical circuit (the circuit with the longest time per token in the CMG). Two circuits are coupled when they have a transition in common. The graph of this section is from a class that has its input circuit "uncoupled" from the critical circuit, i.e., they are connected through other circuits in the graph. The graph of section 4.1 is not as sensitive to variations in the time of the critical circuit as the graph of this section. Since this subject is not in the scope of this thesis, there will be no further analysis of these classes.

Without the help of the Simulation and the Analyzer, this adjustment could not be made in such a short period of time. These adjustments sometimes can be predicted, but the Analyzer is a required tool to discover these realization differences in performance.

V.3 Performance Factors

There is a need for an absolute time independent performance factor to classify the graphs by their performance. The absolute time in a given graph is not as critical as the relative amount of time each node has with respect to each other. If each and every transition time in any of the graphs evaluated in this chapter are multiplied by a constant, the resultant graph has the same critical circuit as the former graph. The difference is in the absolute value of the computations. If the appropriate injection rate is applied at the input, the same resource utilization is obtained.

The TBO performance factor (PFTBO) is obtained by

$$PF_{TBO} = \frac{TBO_{LB}}{TBO_m}$$

where TBO_m is the measured TBO of the system.

The TBIO performance factor (PF_{TBIO}) is obtained by

$$PF_{TBIO} = \frac{TBIO_{LB}}{TBIO_m}$$

where $TBIO_m$ is the measured TBIO of the system.

It should be noted that the maximum possible value of these factors is 1.0. The value of the performance factors for the graphs of Sections 4.1 and 4.2 are presented in the Tables 6 and 7, respectively.

VI.0 FURTHER RESEARCH

During the grant period, the ATAMM model was used as the basis for determining analytically bounds for task computational time and system throughput. An operating strategy which achieves optimum time performance was developed. In addition, a new diagnostic tool was developed with which to evaluate performance and functional unit behavior. The diagnostic tool provided monitoring of detailed system operations and the displaying of global system performance indicators and measures.

Continuation of the present effort includes the development of a new multicomputer test bed. The operating system and communication processes are to obey the ATAMM model and to exhibit a completely distributed graph manager operating system. The operating system is to allow for continuously assigned functional units. This system is to be composed of personal computers communicating over a local area network.

The ongoing research has established ATAMM as a viable basis for the specification of data flow multicomputer systems. Further research should proceed in several directions. An outline of these activities is presented below.

1. Fault Tolerance. Due to the inherent nature of the ATAMM model to allow continuous assignment of the functional units, the soft-fail nature of an ATAMM defined multicomputer system is evident in terms of hardware failure. That is, the algorithm may be expected to continue executing, though with degraded performance, with elimination of functional resources. However, additional effort needs to be directed towards recovery strategies associated with error in the data. One applicable method is triple modular redundancy (TMR),

which involves the triplication of the node processes and majority voting. The TMR strategy needs to be investigated with respect to both performance and the preservation of the ATAMM model.

2. An important part of the ATAMM research program is to enhance the understanding of the relationship between performance measures such as TBIO, TBO, and TT with respect to the algorithm graph characteristics and the availability of functional resources. On the basis of recent observations, research is to be directed toward the improvement of the performance measures as a result of modifying the algorithm graph by the addition of nonexecutable features such as control edges and "dummy" nodes. Present investigations suggest that these graph augmentations may alter and improve certain aspects of performance without changing the underlying algorithm.

3. Overhead. Research should be continued toward the refinement of the node marked graph (NMG) representation. This refinement should better model the time associated with communication overhead and other system overhead in relation to node process time. A goal of this modeling would be to determine limits on algorithm decomposition in view of graph complexity and increased communication overhead.

4. Advanced Hardware. An appropriate step in the ATAMM development is the infusion of the processing rules to advanced technology multicomputer hardware for avionic or space-bourne applications. An appropriate environment would include VHSIC technology such as the MIL-STD-1750A processor as the processing element.

5. Theoretical Advancements. So far the ATAMM model has been used

under somewhat restricted conditions. Further research should include such issues as multiple graphs, nonhomogeneous functional units, reliability, fault recovery strategies, and system architecture which takes advantage of the ATAMM model.

VII. REFERENCES

1. P. Treleaven, D. Brownbridge and R. Hopkins. "Data-driven and demand-driven compute architectures," Computing Surveys, vol. 14, pp. 93-143, March 1982.

2. V. Srini, "An architectural comparison of dataflow systems," Computer, pp. 6888, March 1986.

3. W. Rheinbolt, "Report of the panel on future directions in computational mathematics, algorithms, and scientific software," Sponsored by NSF Grant DMS-85-3483, SIAM, 1985.

4. T. Longo, G. Herzog and D. Maxwell, "A fast single chip 1750A CPU and compatible support components in VHSIC-size CMOS technology," Proceedings of the Government Microcircuit Applications Conference, pp. 317-320, 1986.

5. W. Wehner, W. Everhart, S. Shankar and K. Stalsberg, "A VSHIC architecture for highly parallel image understanding," Proceedings of the Government Microcircuit Applications Conference, pp. 117-120, November 1986.

6. M. Sowa and T. Murata, "A data flow computer architecture with program and token memories," IEEE Transactions on Computers, vol. 31, pp. 820-824, September 1982.

7. K. Kavi, B. Buckles and U. Narayan Bhat, "A formal definition of data flow graph models," IEEE Transactions on Computers, vol. 35, pp. 940-948, November 1986.

8. M. Granski, I. Koren and G. Silberman, "The effect of operation scheduling on the performance of a data flow computer," IEEE Transactions on Computers, vol. 36, pp. 1019-1029, September 1987.

9. L. Jamieson, H. Siegel, E. Delp and A. Whinston, "The mapping of parallel algorithms to reconfigurable parallel architectures," Proceedings of Future Directions in Computer Architecture and Software," D. Agrawal Ed., ARO Contract DAAG29-8-D-0100, pp. 147-154, May 1986.

10. J. Peterson, Petri Net Theory and the Modeling of Systems, Englewood Cliffs, N.J.: Prentice-Hall, 1981.

11. T. Murata, "Circuit theoretic analysis and synthesis of marked graphs," IEEE Transactions on Circuits and Systems, vol. CAS-24, vol. 7, pp. 400-405, July 1977.

12. T. Murata, "Modeling and analysis of concurrent systems," Handbook of Software Engineering, C. Vick and C. Ramamoorthy Editors, pp. 39-63, Van Nostrand Reinhold, 1984.

13. S. Seshu, and M. Reed, Linear Graphs and Electrical Networks, Addison-Wesley Publishing Co., Inc., 1961.

14. J. Sifakis, "Performance evaluation of systems using nets," Net Theory and Applications, W. Brauer Editor, pp. 307-319, Springer-Verlag, 1979.

15. C. Ramamoorthy and G. Ho, "Performance evaluation of asynchronous concurrent systems using Petri nets," IEEE Transactions on Software Engineering, vol. 6, pp. 440-449, September 1980.

16. T. Murata, "Synthesis of decision-free concurrent systems for prescribed resources and performance," IEEE Transactions on Software Engineering, vol. 6, pp. 525-530, November 1980.

17. J. W. Stoughton and R. R. Mielke, "Strategies for concurrent processing of complex algorithms," Proceedings on Future Directions in Computer Architecture and Software, Army Research Office, Charleston, SC, May 1986.

18. J. W. Stoughton and R. R. Mielke, "Petri net model for analysis of concurrently processed complex algorithms," Proceedings of IEEE Southeastcon 1986, Richmond, VA, March 1986.

19. J. Stoughton and R. Mielke, "Petri net model for concurrent processing of complex algorithms," Proceedings of the Government Microcircuit Applications Conference, San Diego, California, pp. 11-14, November 1986.

20. T. Murata, "Use of resource-time product concept to derive a performance measure of timed petri nets," Proceedings of Midwest Symposium on Circuits and Systems, Vol. 28, pp. 407-410, August 1985.

21. K. Jackson, R. Tymchyshyn, R. Mielke and J. Stoughton, "Simulation software for concurrent processing," Proceedings of the IEEE Southeastcon, Tampa, Florida, pp. 82-86, April 1987.

TABLES

GRAPH WITH PARALLEL PATHS

PRIORITY ASSIGNMENT 5 4 3 2 2 1

RESOURCES	COMPUTING POWER	RESOURCE UTILIZATION	AVERAGE TBQ	AVERAGE TBIO	MAXIMUM CONCURRENCY
1	46210	99.11%	3243.0	3235.0	1
2	49484	97.49%	1627.5	2592.0	2
3	49409	90.07%	1307.0	2265.0	3
4	49856	68.55%	1136.5	2265.0	4
5	49355	57.31%	1083.0	2265.0	5
6	49355	47.76%	1083.0	2265.0	5
7	49355	40.93%	1083.0	2265.0	5

TABLE 1. Results from first experiment, first priority assignment.

GRAPH WITH PARALLEL PATHS

PRIORITY ASSIGNMENT 1 2 2 3 4 5

RESOURCES	COMPUTING POWER	RESOURCE UTILIZATION	AVERAGE TBQ	AVERAGE TBIO	MAXIMUM CONCURRENCY
1	49818	99.11%	3243.0	3235.0	1
2	50068	97.67%	1623.0	2592.0	2
3	49486	90.70%	1294.9	2265.0	3
4	49836	68.50%	1136.0	2265.0	4
5	50002	57.34%	1083.0	2265.0	5
6	50002	47.78%	1083.0	2265.0	5
7	50002	40.96%	1083.0	2265.0	5

TABLE 2. Results from first experiment, second priority assignment.

GRAPH WITH PARALLEL PATHS

PRIORITY ASSIGNMENT 1 7 2 6 3 4 5

RESOURCES	COMPUTING POWER	RESOURCE UTILIZATION	AVERAGE TBO	AVERAGE TBIO	MAXIMUM CONCURRENCY
1	49818	93.11%	3243.0	5838.0	1
2	50081	97.73%	1622.0	3243.0	2
3	43511	80.72%	1324.0	2346.8	3
4	49892	68.56%	1137.0	2273.0	4
5	49999	57.35%	1083.0	2273.0	5
6	49999	47.79%	1083.0	2273.0	5
7	49999	40.97%	1083.0	2273.0	5

TABLE 3. Results from first experiment, third priority assignment.

GRAPH WITH ITERATIVE LOOPS

PRIORITY 4 3 2 1 6 5

RESOURCES	COMPUTING POWER	RESOURCE UTILIZATION	AVERAGE TBI	AVERAGE TBO	AVERAGE TBIO	MAXIMUM CONCURRENCY
1	37616	99.16%	2594.0	2594.0	2589.0	1
2	38571	97.24%	1294.0	1301.0	1614.0	2
3	39211	85.61%	964.0	975.0	1679.0	3
4	39211	64.20%	964.0	975.0	1679.0	3
5	39211	51.36%	964.0	975.0	1679.0	3
6	39211	42.80%	964.0	975.0	1679.0	3

Table 4. Results from second experiment, first priority assignment.

GRAPH WITH ITERATIVE LOOPS

PRIORITY 5 6 1 2 3 4

RESOURCES	COMPUTING POWER	RESOURCE UTILIZATION	AVERAGE TBI	AVERAGE TBO	AVERAGE TBIO	MAXIMUM CONCURRENCY
1	40186	99.16%	2594.0	2594.0	5938.0	1
2	39319	97.54%	1294.0	1300.3	1987.9	2
3	39261	85.97%	961.0	971.0	1671.5	3
4	39261	64.48%	961.0	971.0	1671.5	3
5	39261	51.58%	961.0	971.0	1671.5	3
6	39261	42.98%	961.0	971.0	1671.5	3

Table 5. Results from second experiment, second priority assignment.

PERFORMANCE FACTORS FOR GRAPH WITH PARALLEL PATHS

Resources	PF_{TBO}	PF_{TBIO}
1	0.328399	0.692426
2	0.654579	0.867544
3	0.914843	0.988962
4	0.937500	0.988962
5	0.963379	0.988962
6	0.963379	0.988962
7	0.963379	0.988962

Table 6. Performance factors for graph of Section 4.1.

PERFORMANCE FACTORS FOR GRAPH WITH ITERATIVE LOOPS

Resources	PF_{TBO}	PF_{TBIO}
1	0.370084	0.617999
2	0.737893	0.991325
3	0.984615	0.987654
4	0.984615	0.987654
5	0.984615	0.987654
6	0.984615	0.987654

Table 7. Performance factors for graph of Section 4.2.

FIGURES

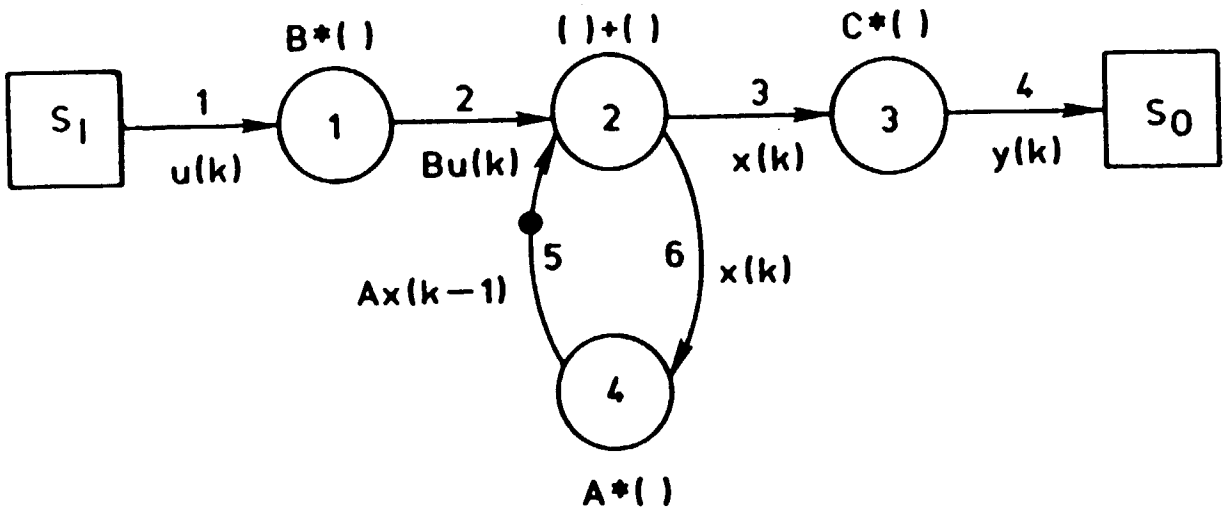
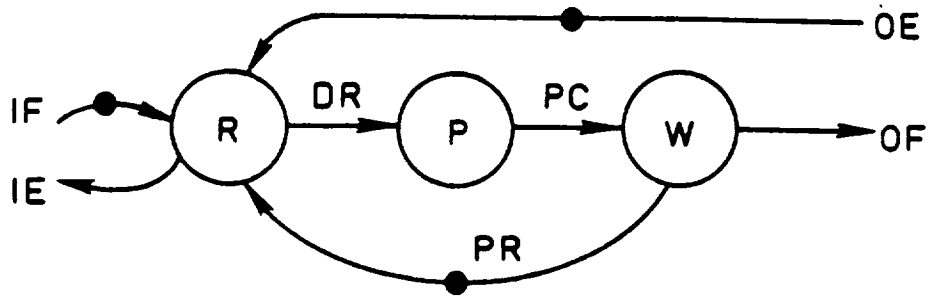


Figure 1. Algorithm marked graph for discrete system equation.



NMG EDGE LABELS

IF	Input Buffer Full
IE	Input Buffer Empty
DR	Data Read
PC	Process Complete
PR	Process Ready
OE	Output Buffer Empty
OF	Output Buffer Full

Figure 2. ATAMM node marked graph model.

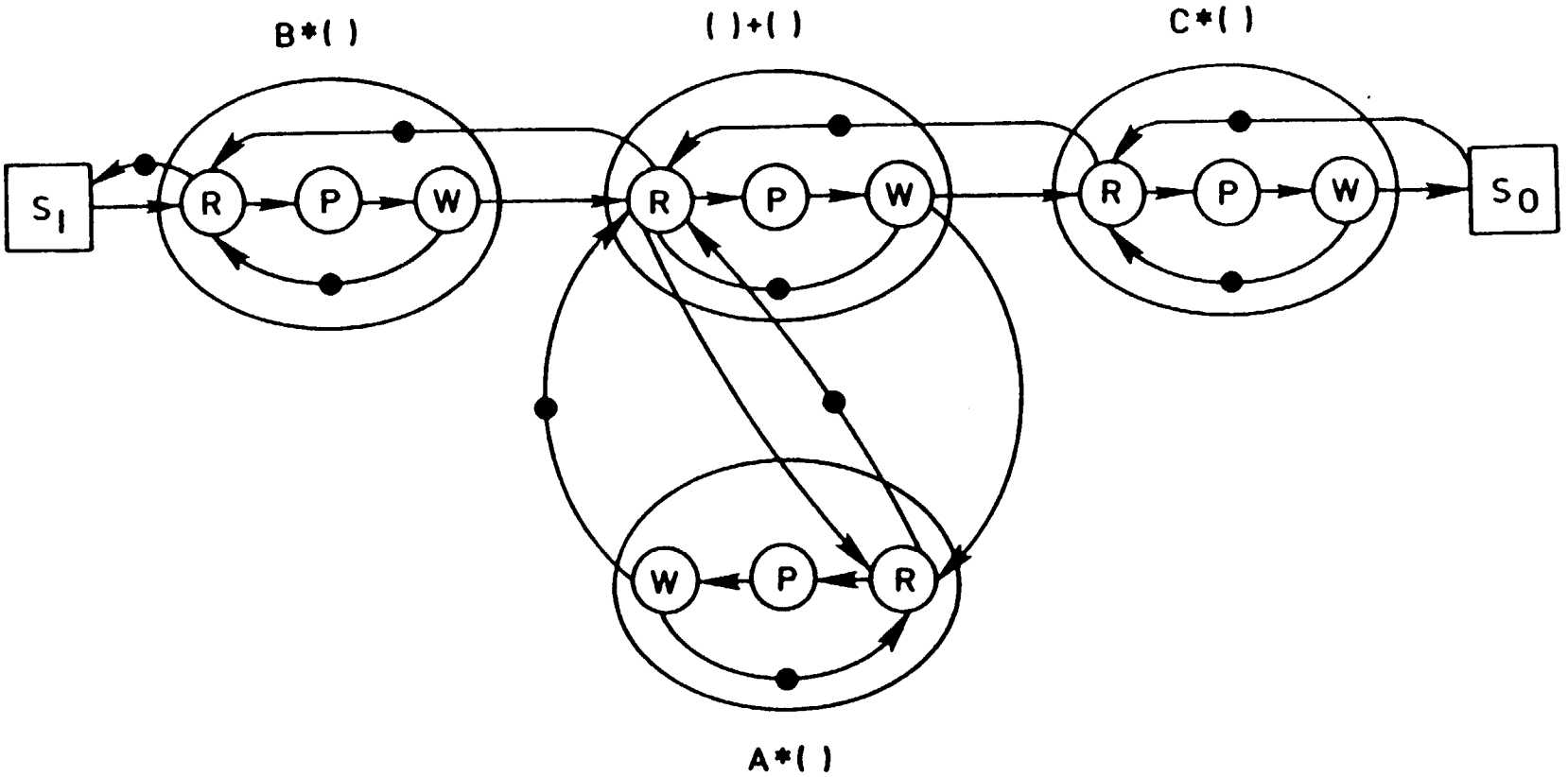


Figure 3. ATAMM computational marked graph model for discrete system equation.

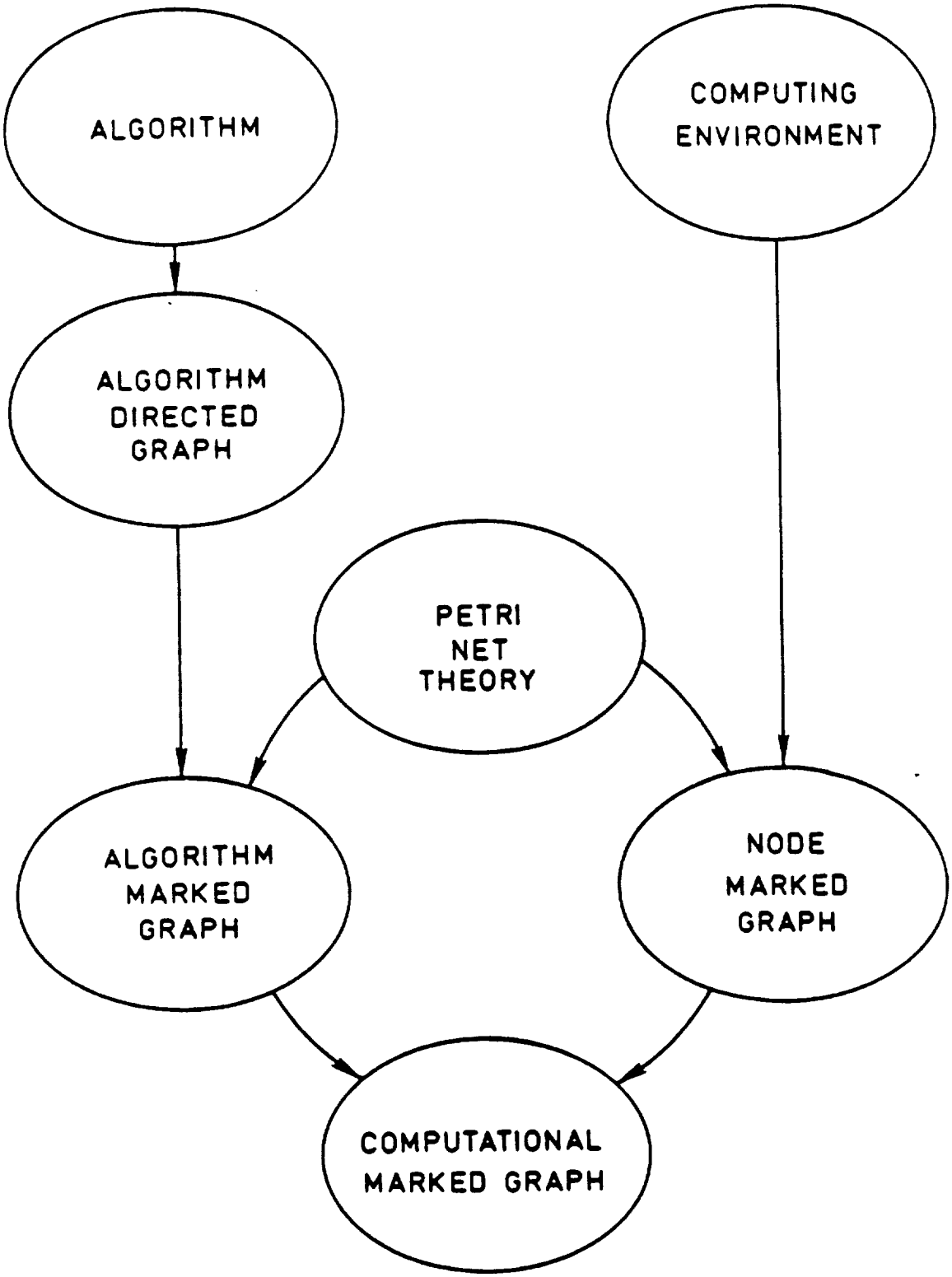


Figure 4. ATAMM model components.

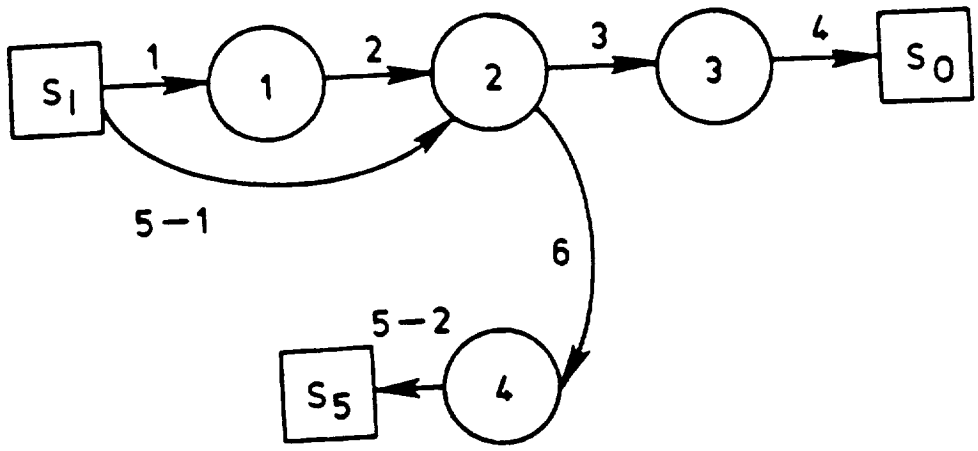


Figure 5. Modified algorithm graph for Figure 1.

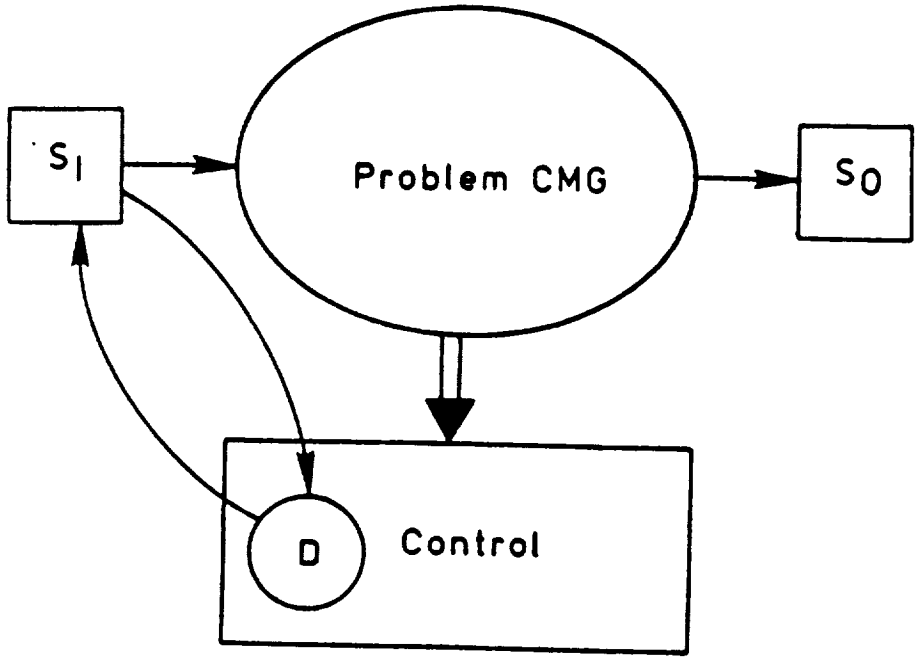


Figure 6. Operating strategy implementation.

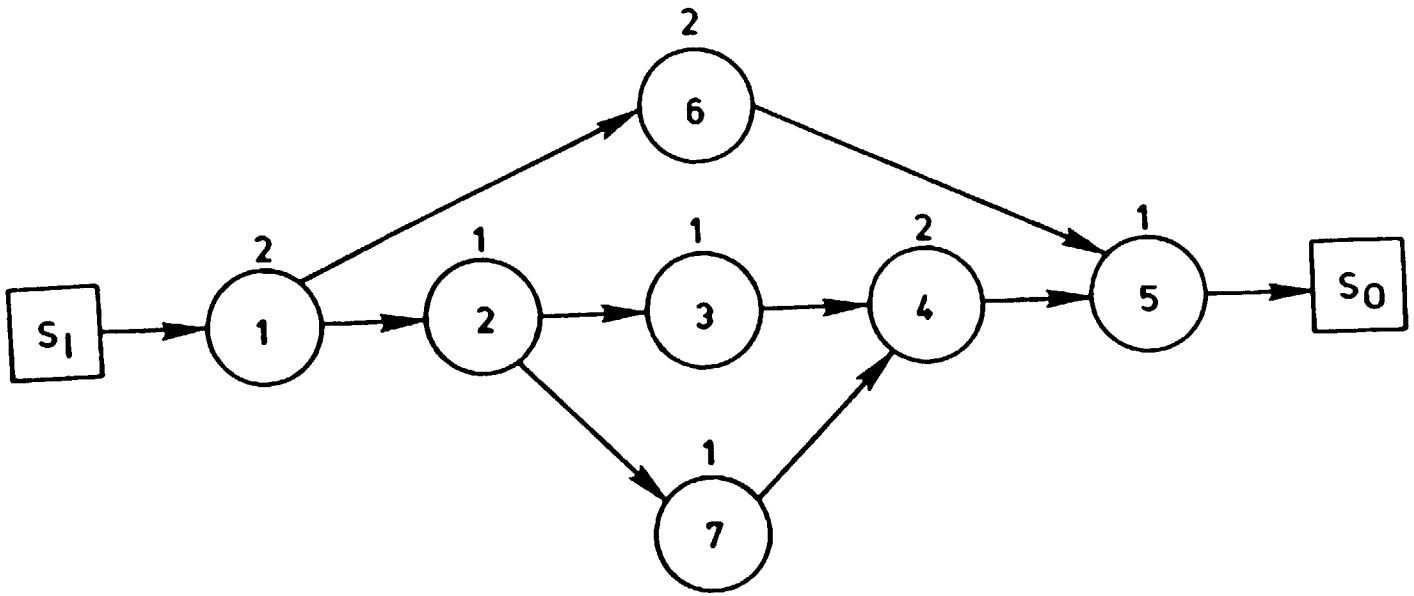


Figure 7. Algorithm graph for design example.

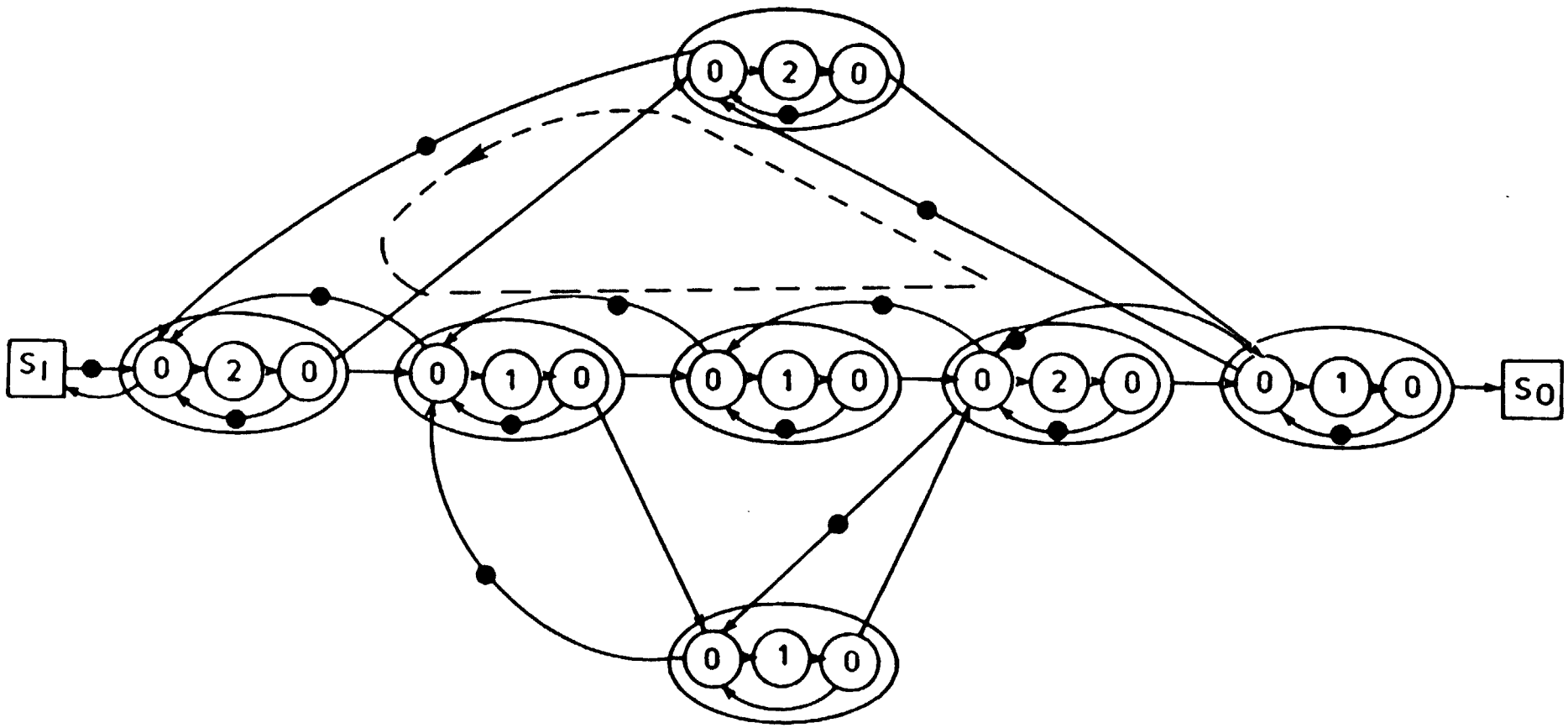


Figure 8. Computational marked graph for design example.

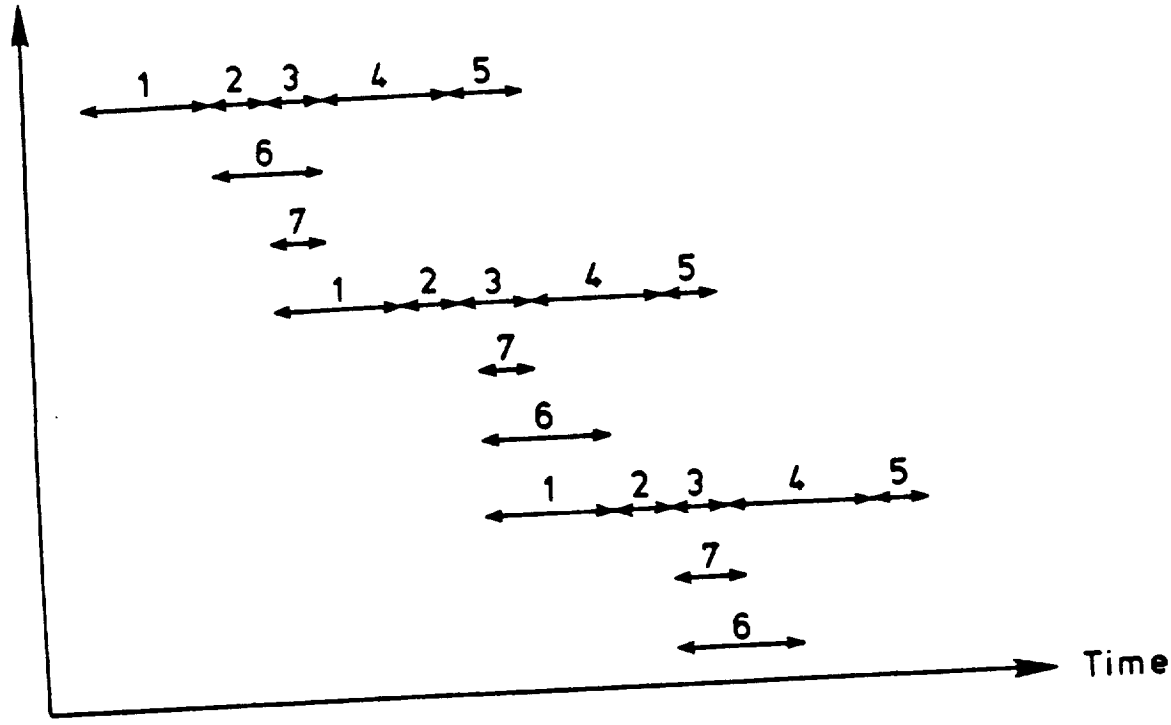


Figure 9. Graph play with TBO=3 and unlimited functional units.

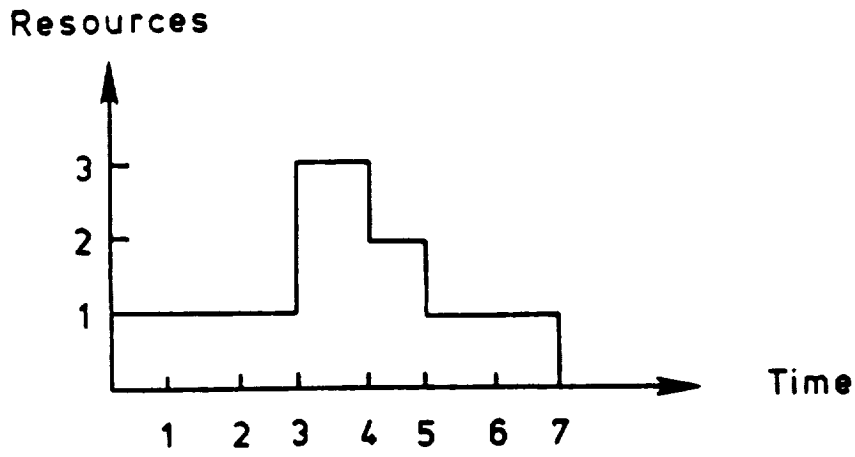


Figure 10. Resource utilization envelope for design example.

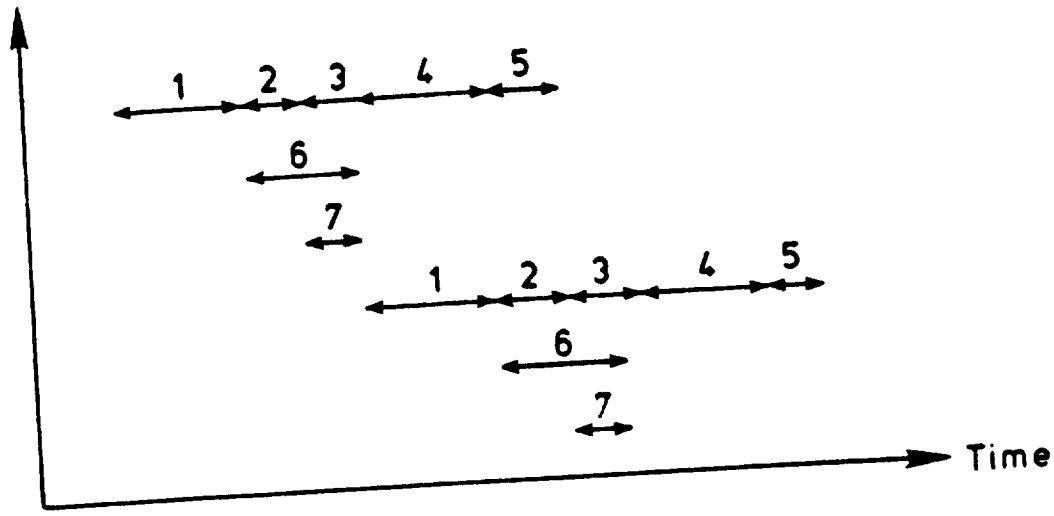


Figure 11. Graph play with TBO=4 and no control edges.

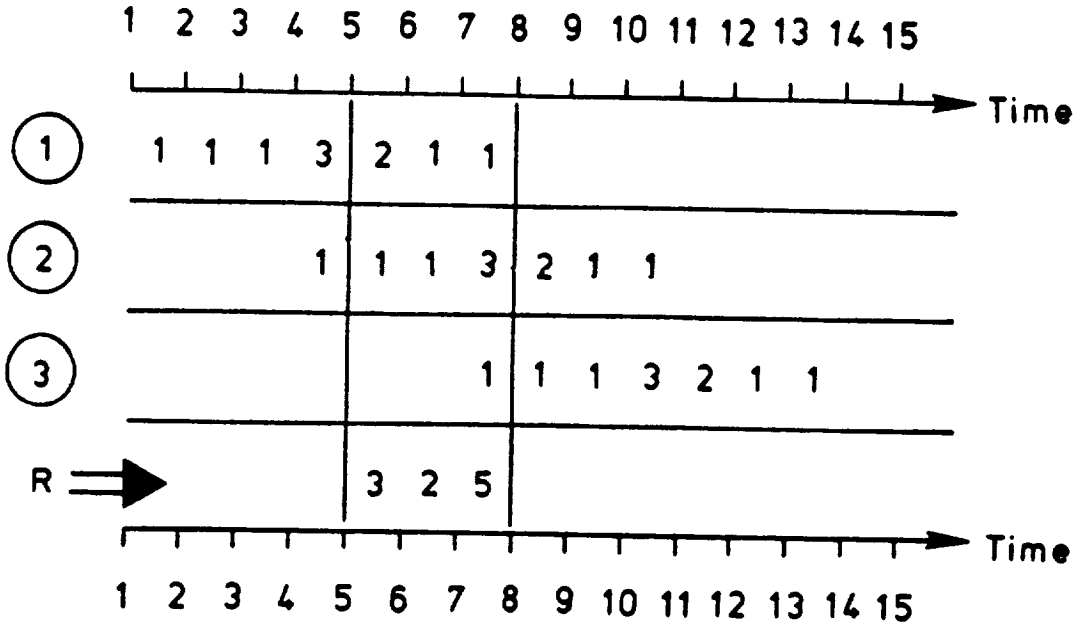


Figure 12. Resource envelope overlay diagram with TBO=3.

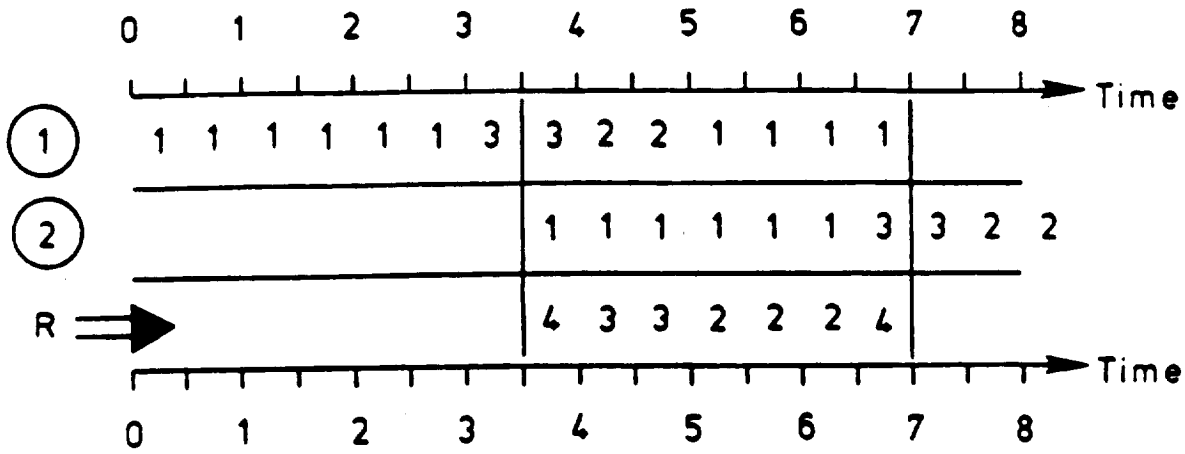


Figure 13. Resource envelope overlay diagram with TBO=3.5.

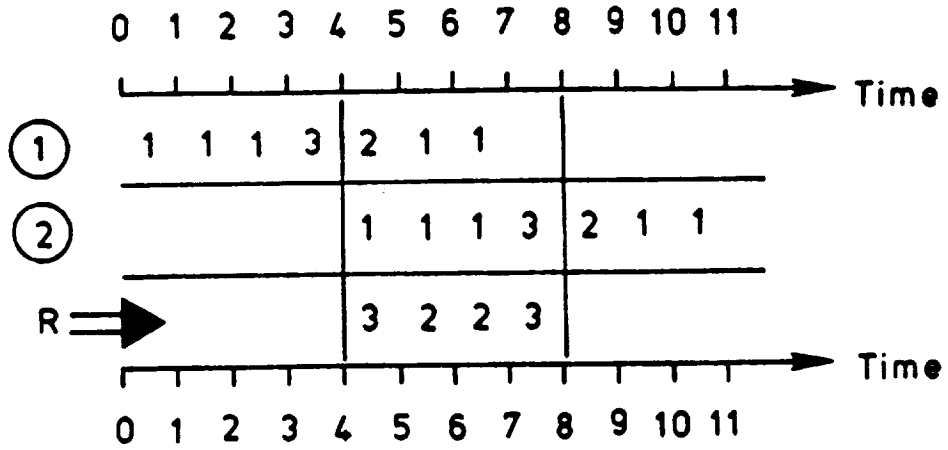


Figure 14. Resource envelope overlay diagram with TBO=4.0.

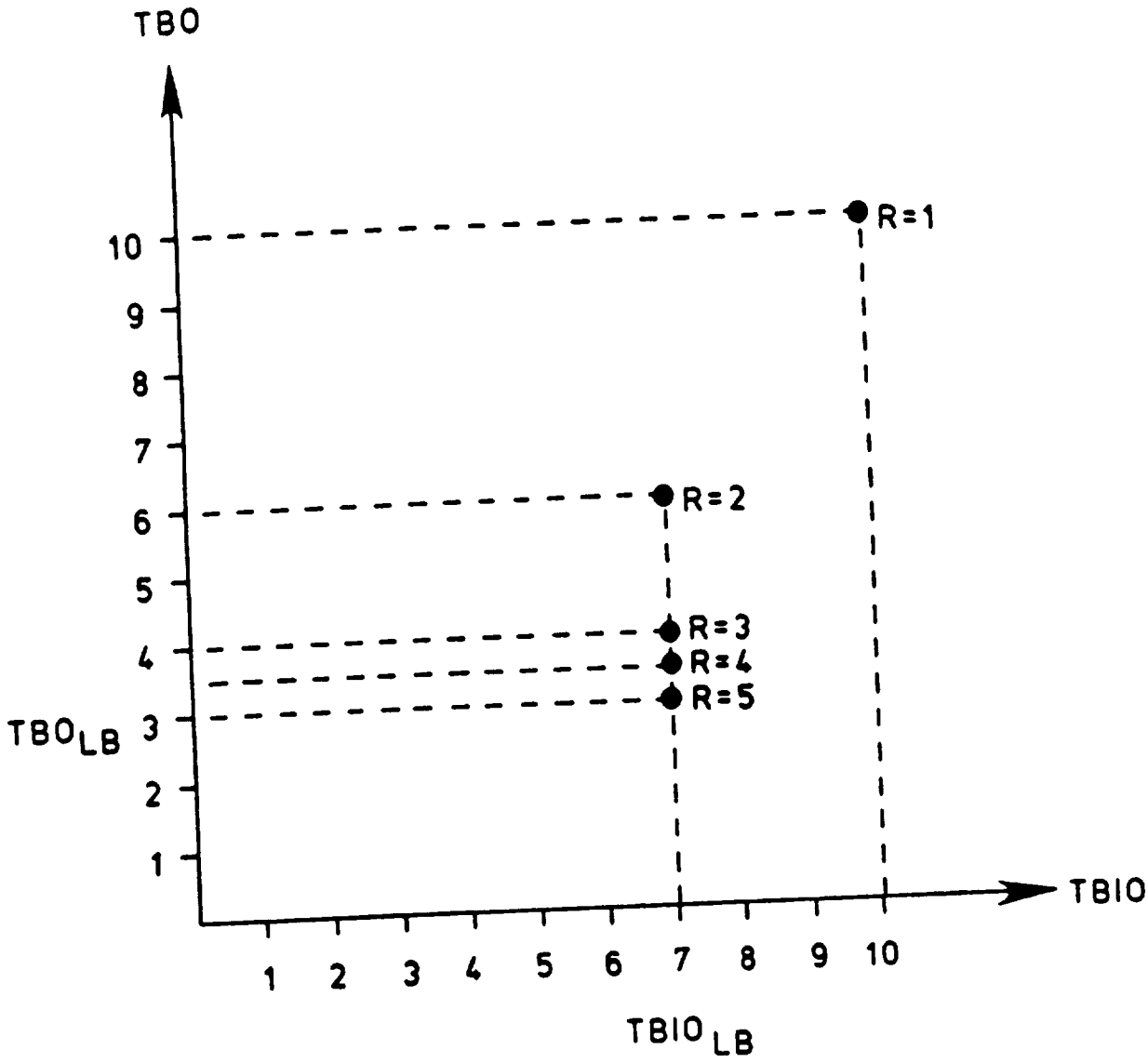


Figure 15. Example algorithm graph performance analysis summary.

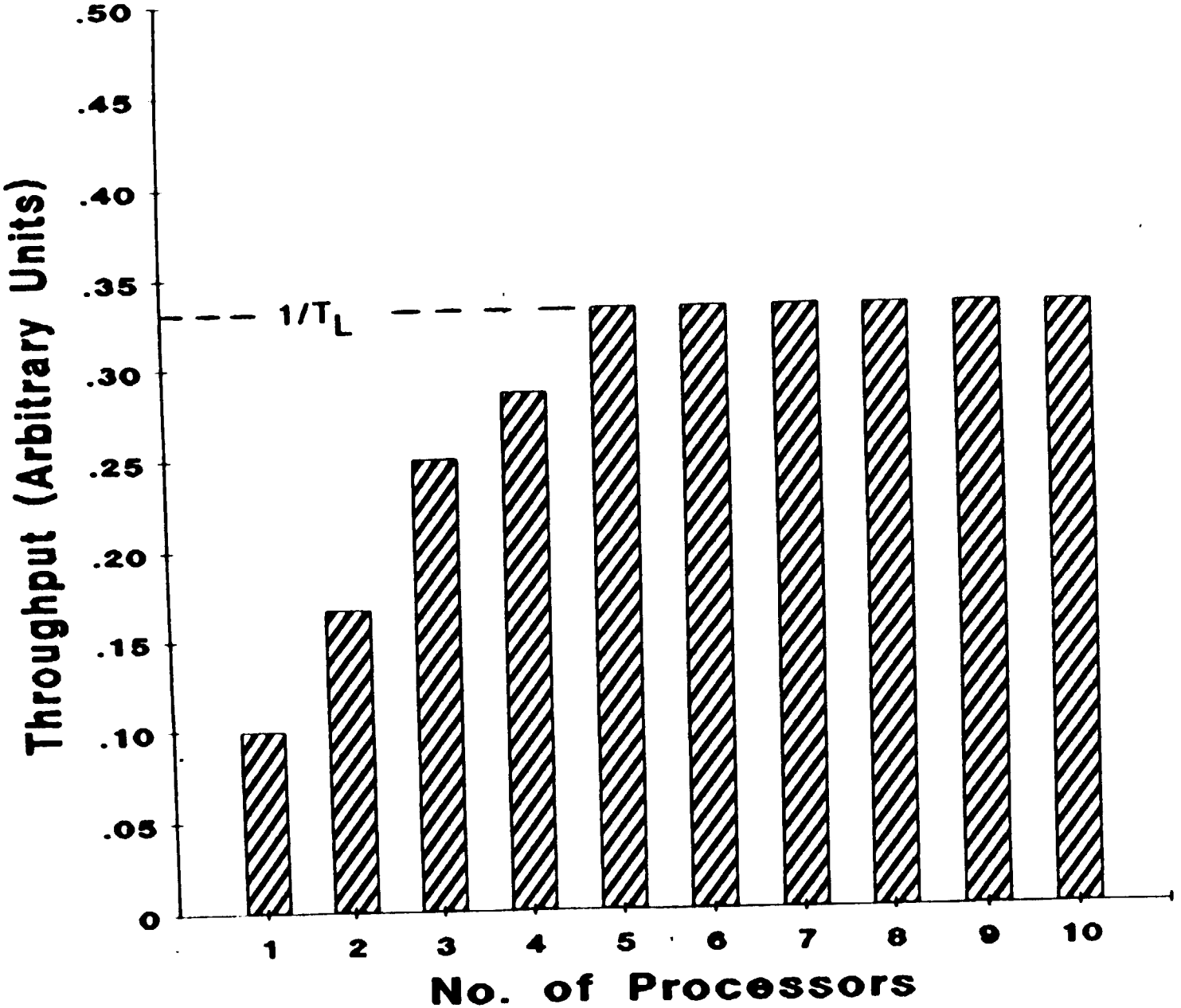


FIGURE 16. PERFORMANCE MARGIN FOR EXAMPLE ALGORITHM.

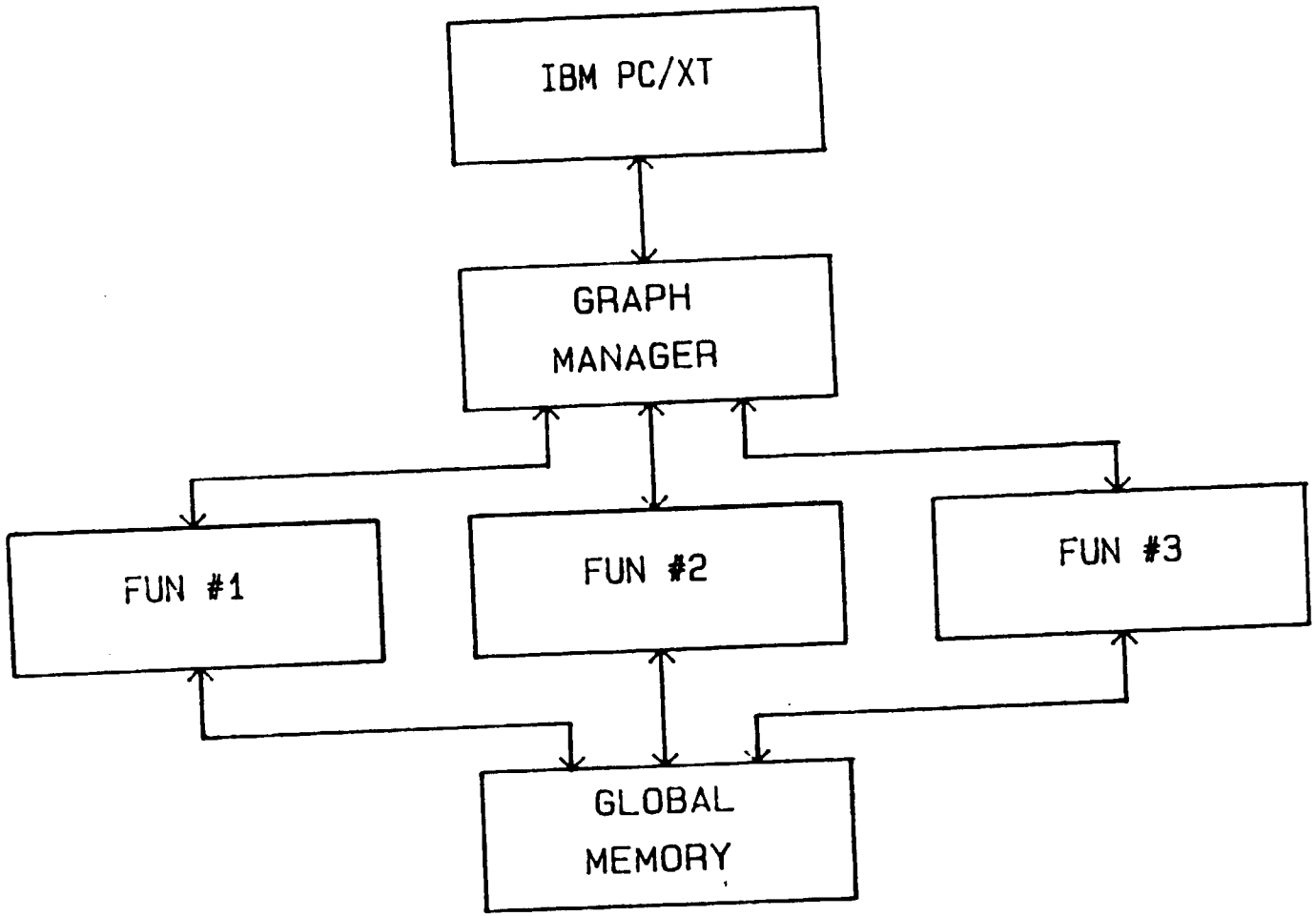


Figure 17 Prototype block diagram.

Figure 18. Prototype communications dialog.

FIPSO file	Word description
T,72,N,1,F,1	<-- Node 1 is fired at time 72 and assigned to FU1
T,116,N,1,I	<-- Node 1 read the input places at time 116
T,249,N,1,P	<-- Node 1 finished the process at time 249
T,250,N,1,S	<-- Node 1 got clearance to output data at time 250
T,276,N,1,O	<-- Node 1 wrote the output data at time 276
T,278,N,2,F,2	<-- Node 2 is fired at time 278 and assigned to FU2
T,322,N,2,I	<-- Node 2 read the input places at time 322
T,323,N,1,F,3	<-- Node 1 is fired at time 323 and assigned to FU3
T,367,N,1,I	<-- Node 1 read the input places at time 367
T,455,N,2,P	<-- Node 2 finished the process at time 455
T,456,N,2,S	<-- Node 2 got clearance to output data at time 456
T,482,N,2,O	<-- Node 2 wrote the output data at time 482
T,485,N,3,F,4	<-- Node 3 is fired at time 485 and assigned to FU4
T,500,N,1,P	<-- Node 1 finished the process at time 500
T,501,N,1,S	<-- Node 1 got clearance to output data at time 501
T,527,N,1,O	<-- Node 1 wrote the output data at time 527
T,661,N,3,I	<-- Node 3 read the input places at time 661
T,663,N,2,F,5	<-- Node 2 is fired at time 663 and assigned to FU5
T,707,N,2,I	<-- Node 2 read the input places at time 707
T,708,N,1,F,1	<-- Node 1 is fired at time 708 and assigned to FU1
T,752,N,1,I	<-- Node 1 read the input places at time 752
T,840,N,2,P	<-- Node 2 finished the process at time 840
T,841,N,2,S	<-- Node 2 got clearance to output data at time 841
T,867,N,2,O	<-- Node 2 wrote the output data at time 867
T,885,N,1,P	<-- Node 1 finished the process at time 885
T,886,N,1,S	<-- Node 1 got clearance to output data at time 886
T,912,N,1,O	<-- Node 1 wrote the output places at time 912
T,1190,N,3,P	<-- Node 3 finished the process at time 1190
T,1191,N,3,S	<-- Node 3 got clearance to output data at time 1191
T,1295,N,3,O	<-- Node 3 wrote the output places at time 1295

Figure 19. A sample FIPSO file.

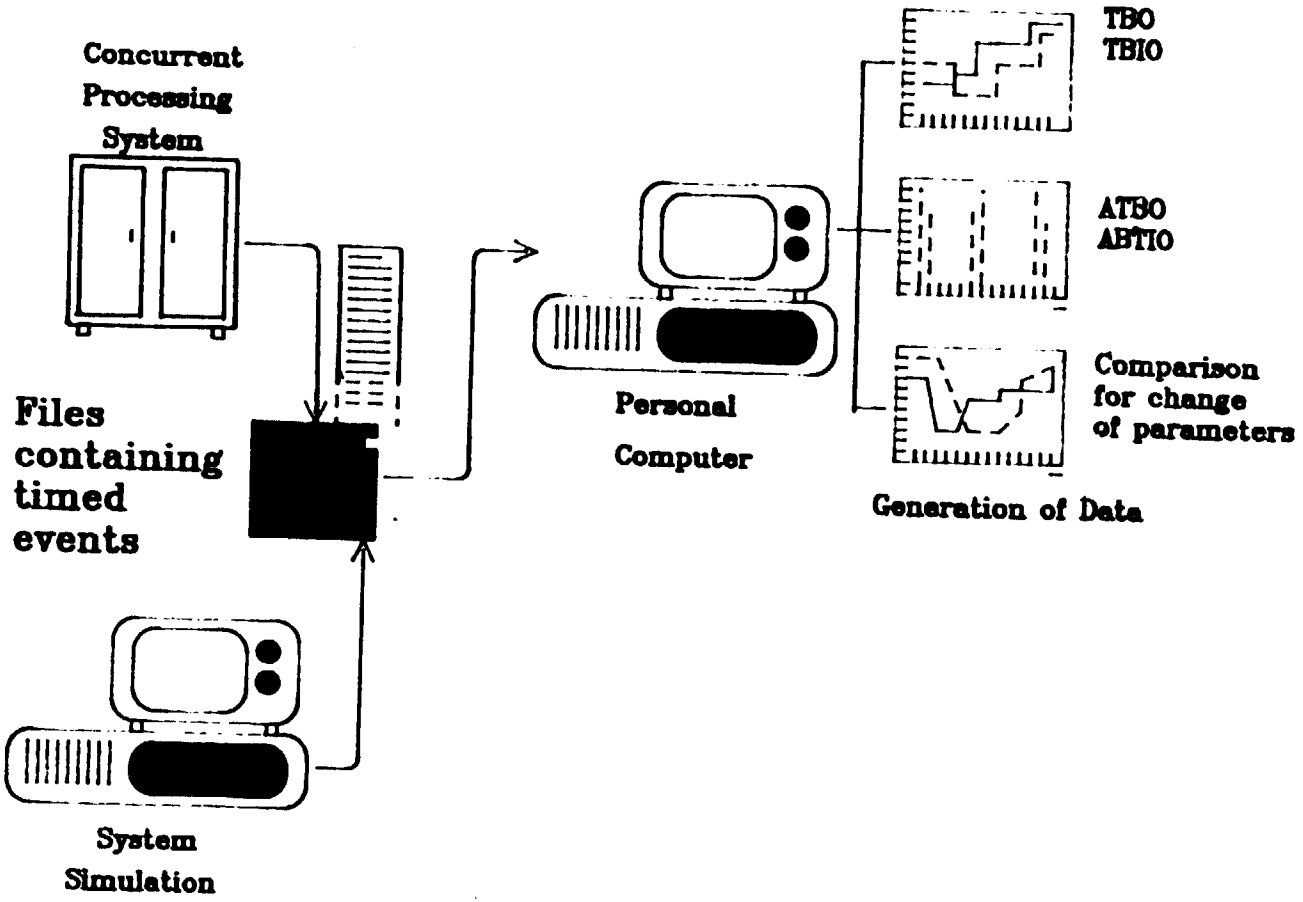


Figure 20. Analyzer information flow.

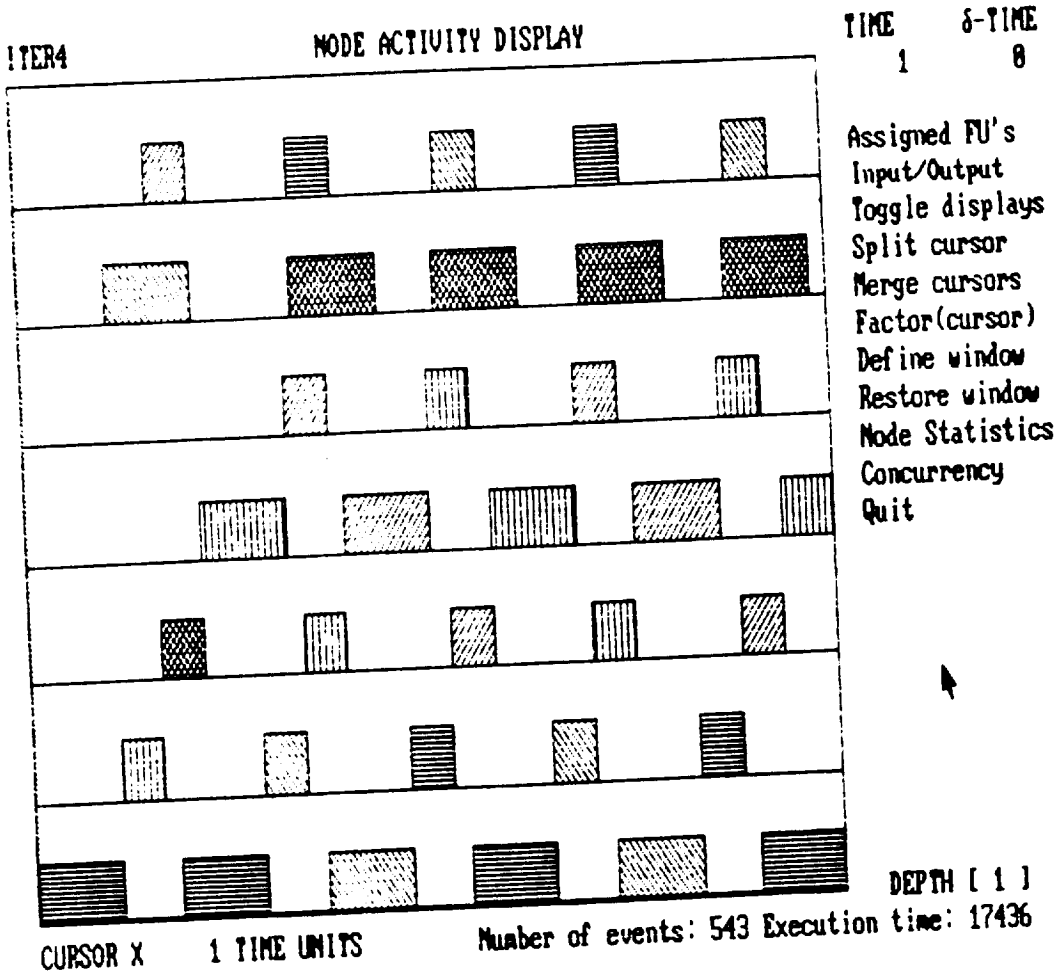


Figure 21. Analyzer Node Activity Display.

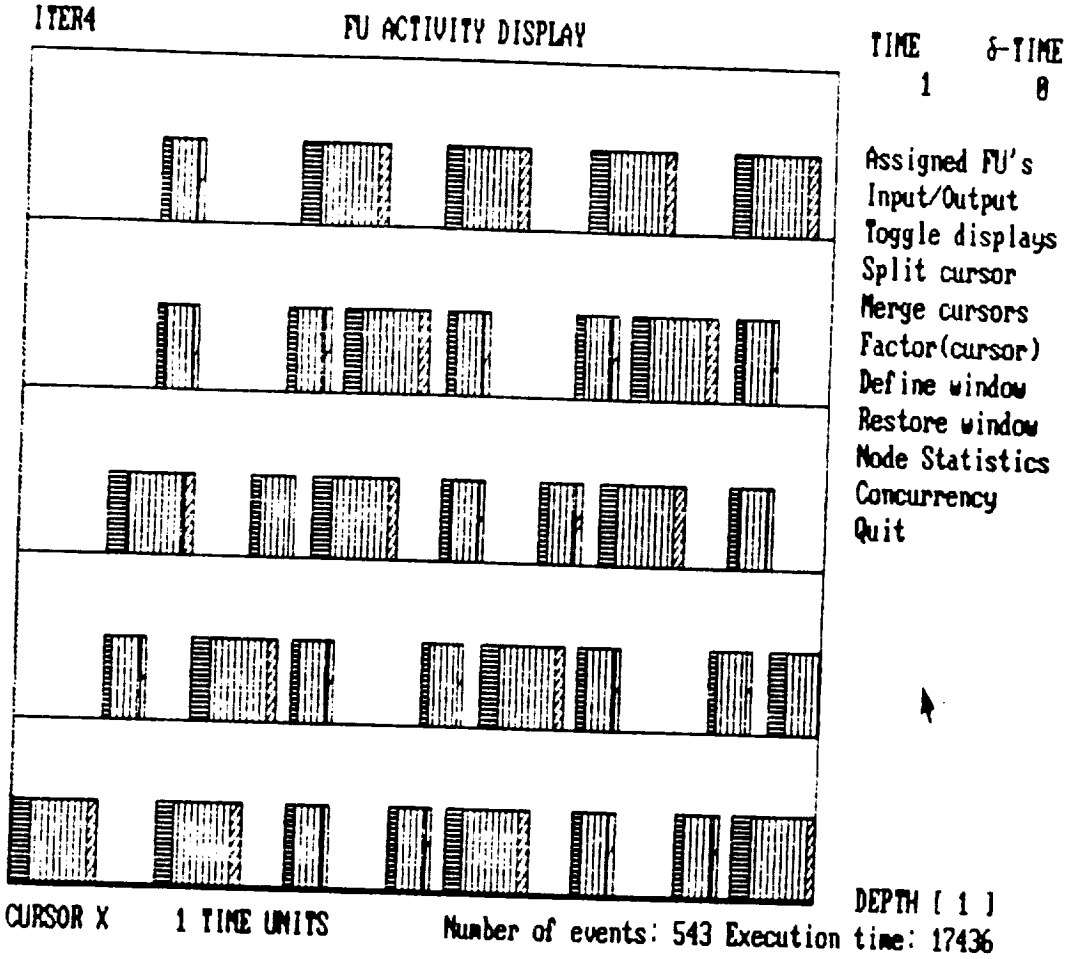


Figure 22. Analyzer Functional Unit Display.

ORIGINAL PAGE IS
OF POOR QUALITY

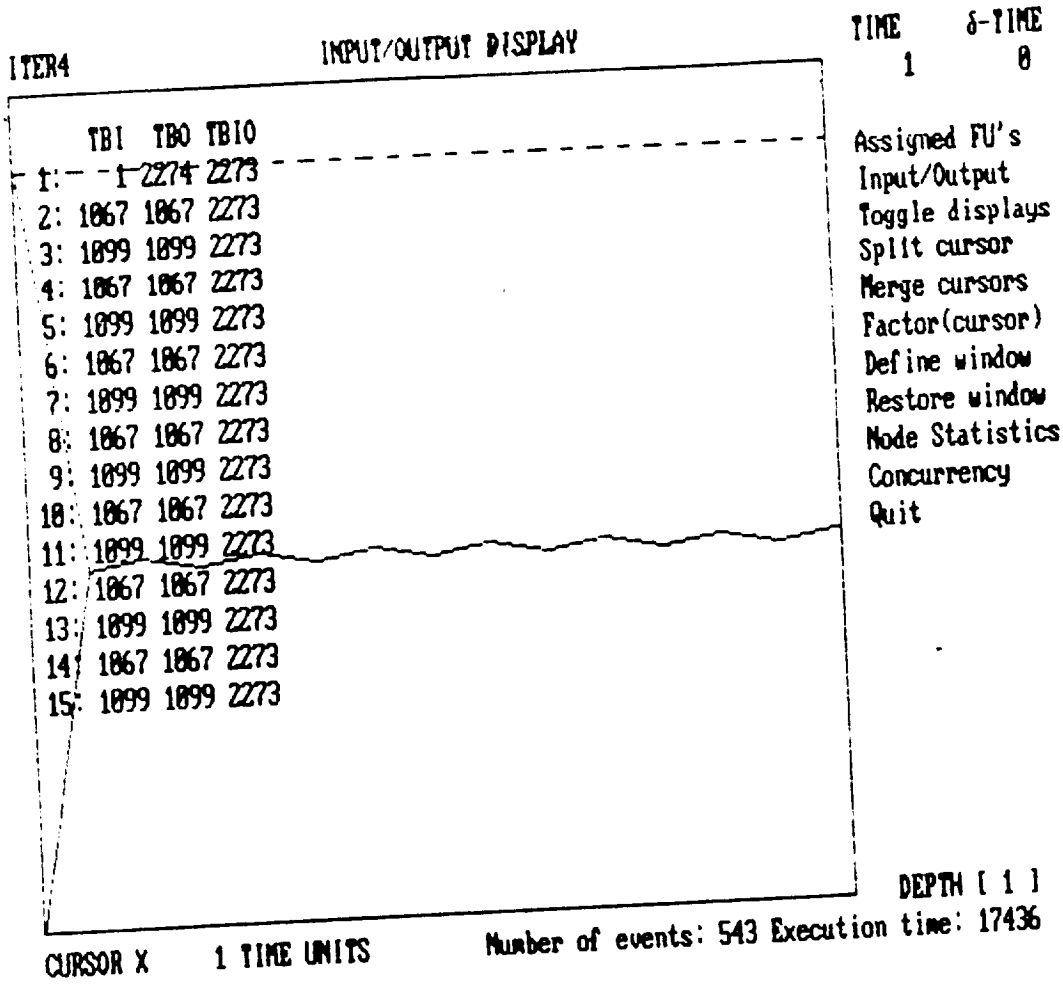


Figure 23. Analyzer Input/Output Display.

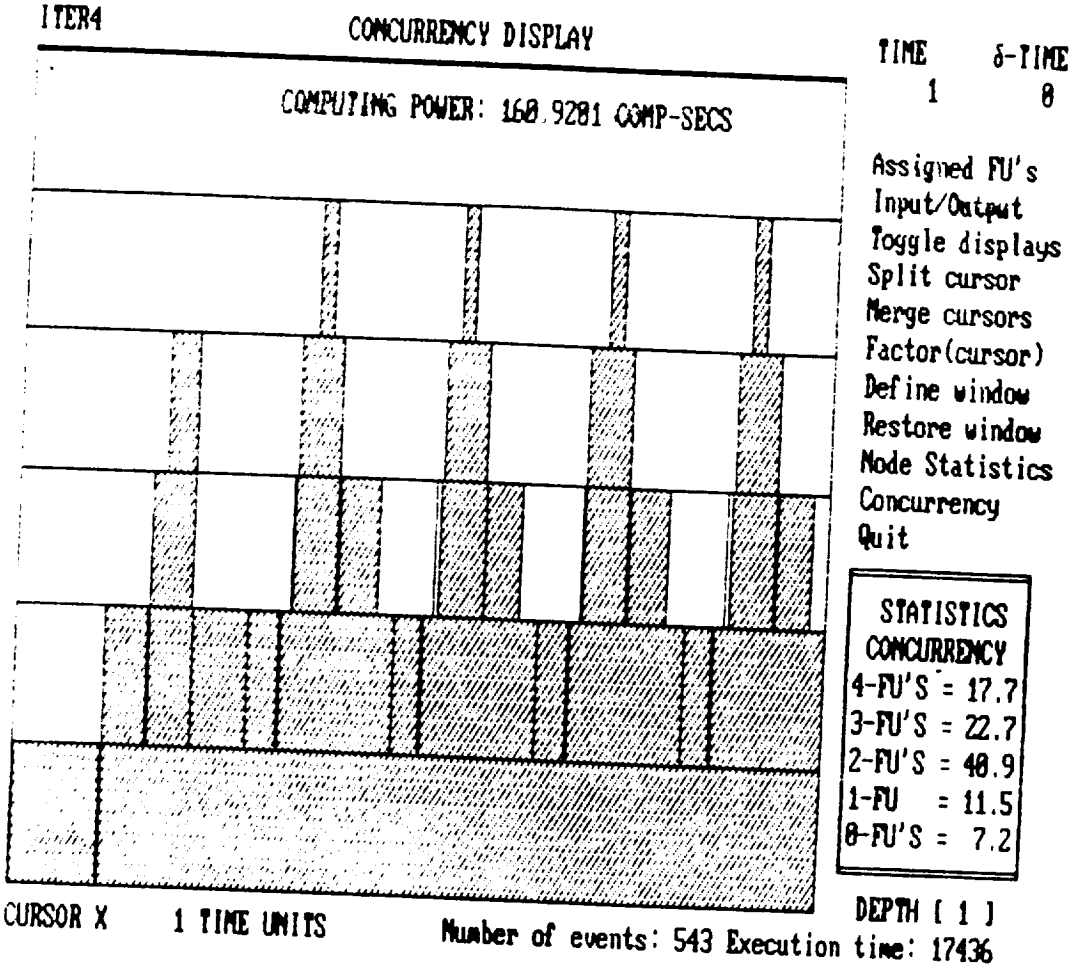


Figure 24. Analyzer Concurrency Display.

C-2

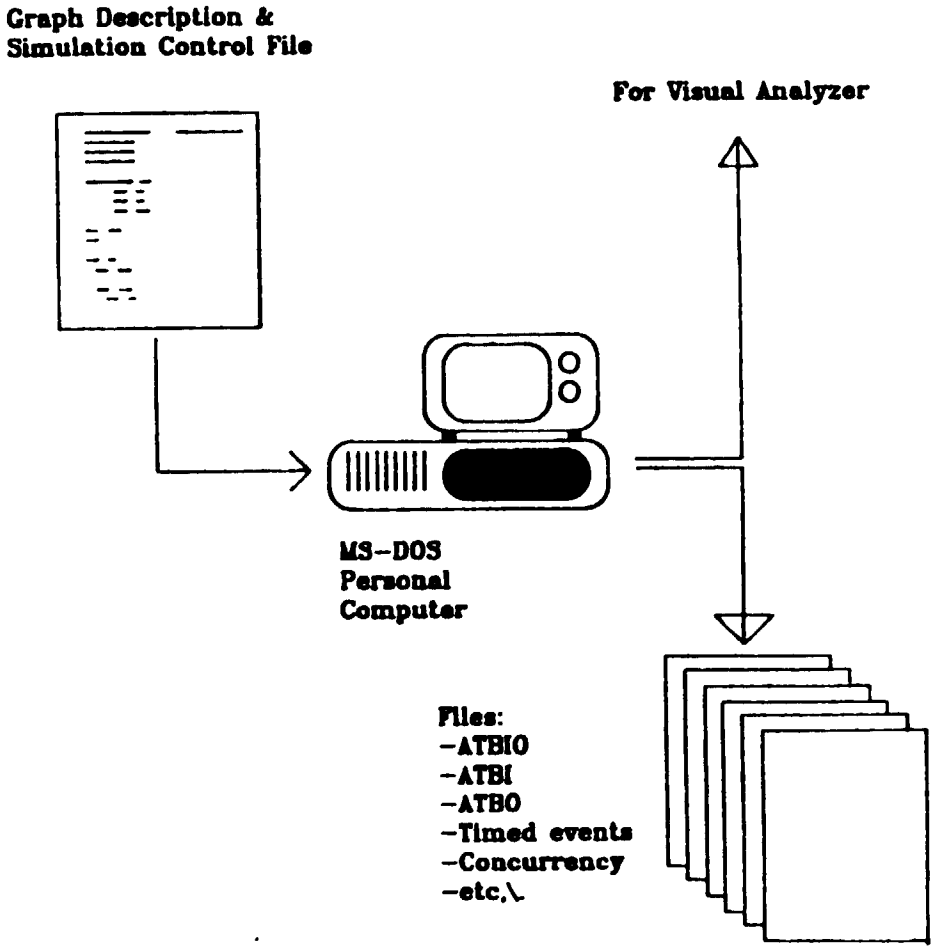


Figure 25. Graph Simulation/Analyzer information flow.

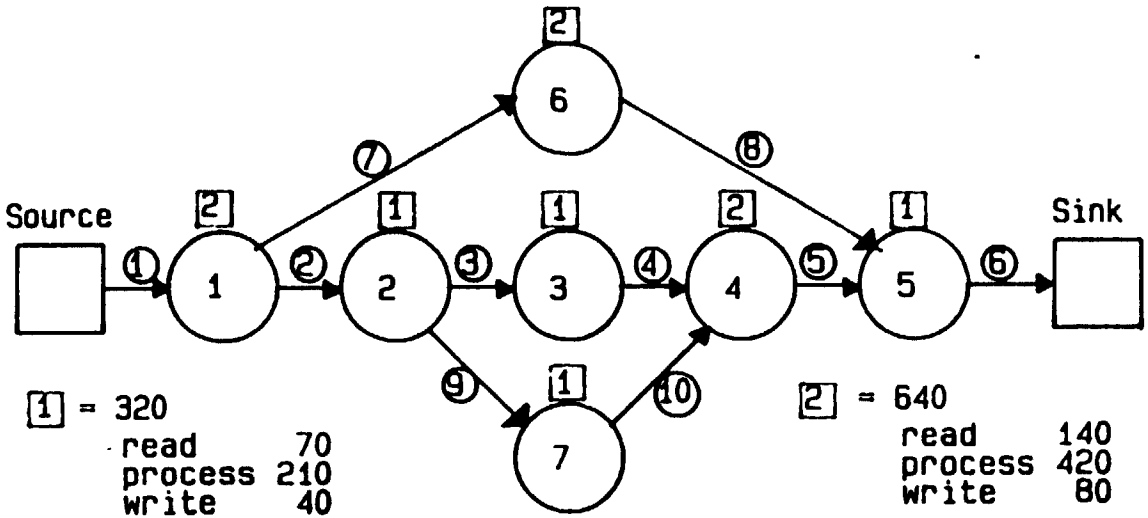


Figure 26. Graph with parallel paths.

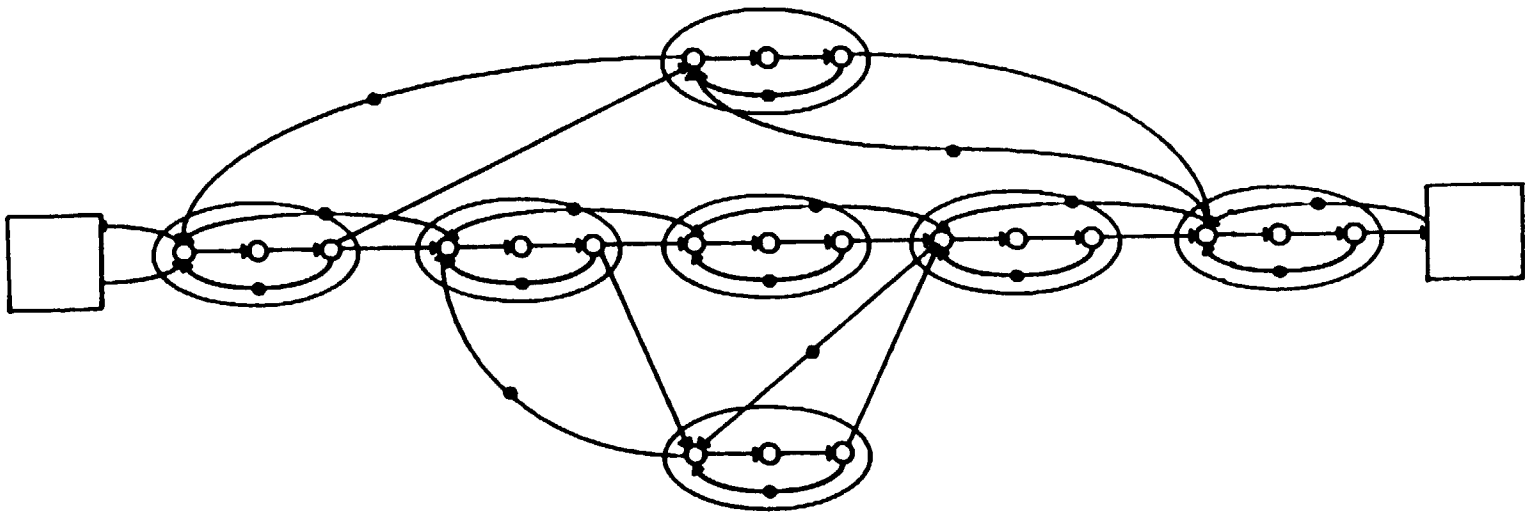


Figure 27. CMG using Single Node model.

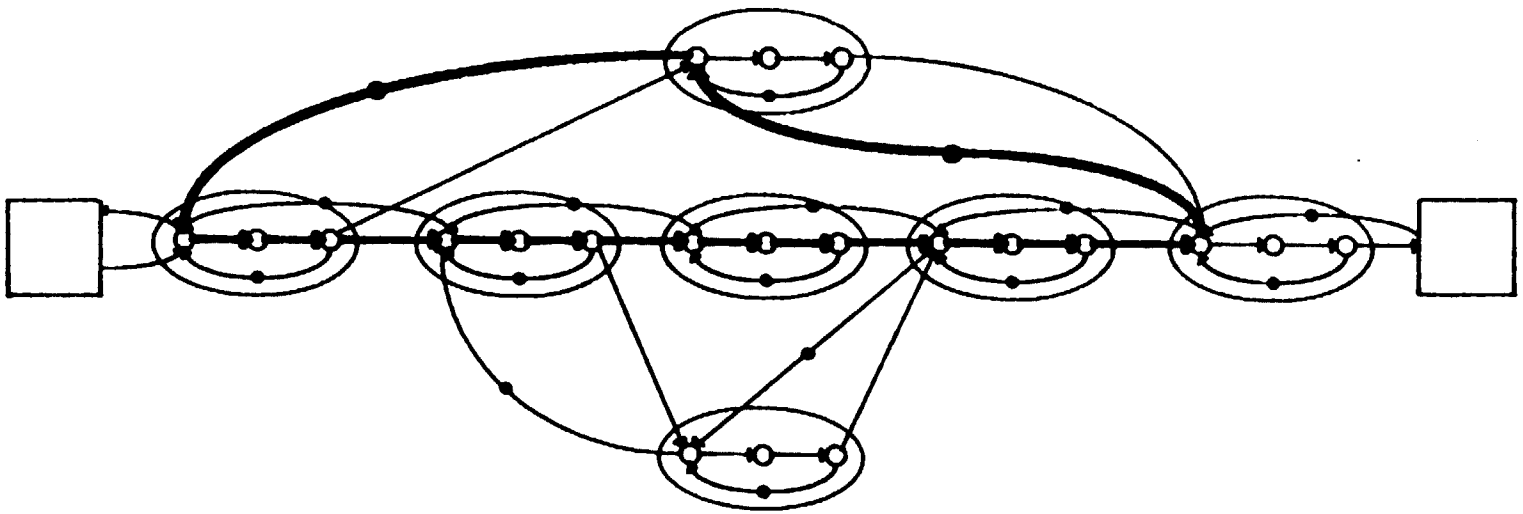


Figure 28. Circuit to obtain T_{LB}

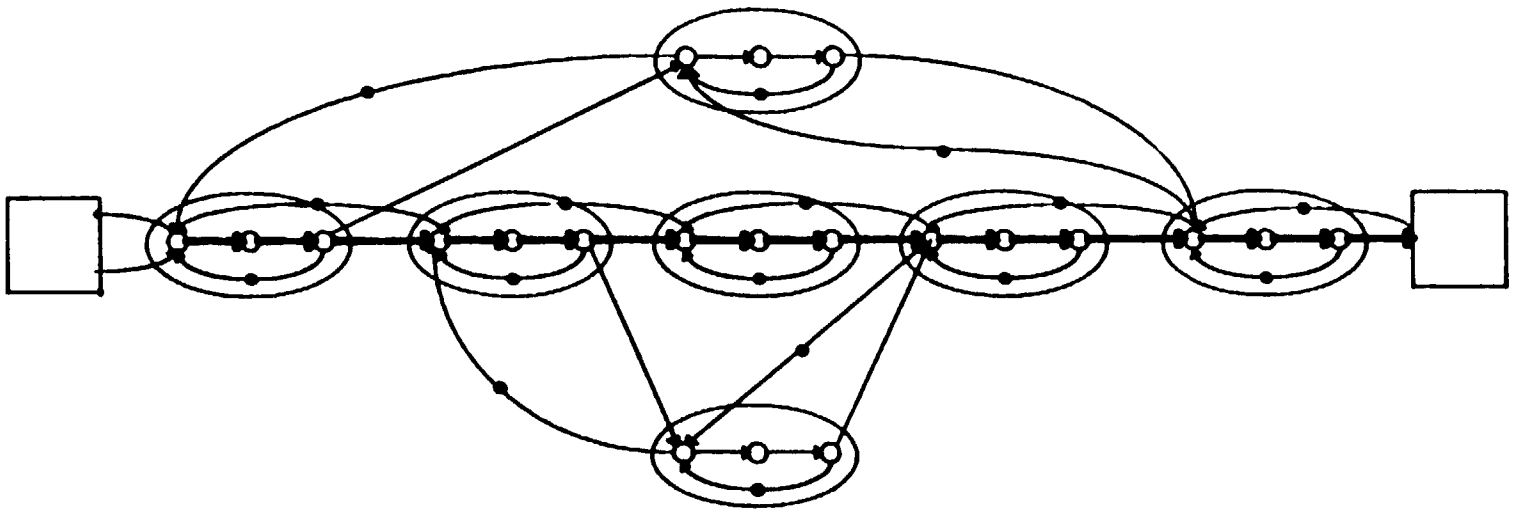


Figure 29. Path to obtain $TBIO_{1LB}$.

```
# *** Simulation of a graph with parallel paths ***
# *** is simulated with full range of resources ***
# *** and with two different priority assignments ***

Graph Graph with parallel paths  TBO_LB = 1065  TBIO_LB = 2240
    Nodes 7
    Sources 1
    Sinks 1
    Places 10
    Resources -1                    # *** From 1 to 7 resources ***
    Priority 5 4 3 7 2 6 1     # Alternate assignment 1 6 2 7 3 4 5
    Tokens 1                        # Data available at the input node
    Model Single
    Input 1                     # The input node is node 1
    Output 5                    # The output node is node 5
    Times               # Global time assignment
        Read    70              # These time assignments are for all
        Process 210         # nodes in the graph. They can be
        write   40              # overridden later on.

    Node 1
    Inputs 1
    Outputs 2 7
    Time                # Local time assignment
        Read   140              # These time assignments overide
        Process 420         # the global time assignments
        write   80              #

    Node 2
    Inputs 2
    Outputs 3 9

    Node 3
    Inputs 3
    Outputs 4

    Node 4
    Inputs 4 10
    Outputs 5
```

Figure 30. Graph Description and Simulation Control file used for the first experiment.

```
Time                # Local time assignment
     Read   140             # These time assignments override
     Process 420          # the global time assignments
     write   80             #

Node 5
Inputs 5 8
Outputs 6

Node 6
Inputs 7
Outputs 8
Time                # Local time assignment
     Read   140             # These time assignments override
     Process 420          # the global time assignments
     Write   80             #

Node 7
Inputs 9
Outputs 10

Source 1
Outputs 1
Time
     Write 925    # Source output write time is TBO_LB - IN1
                  # Write time = 1065 - 140

Sink 2
Inputs 6
Time
     Read 70

End                 # This ends the Graph Description File
```

Figure 30. (Continuation).

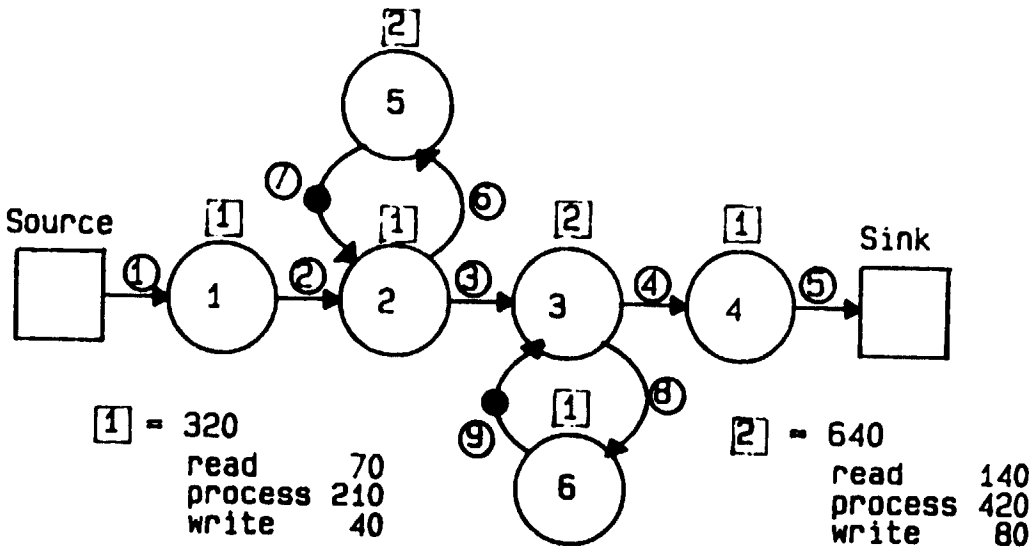


Figure 31. Graph with iterative loops.

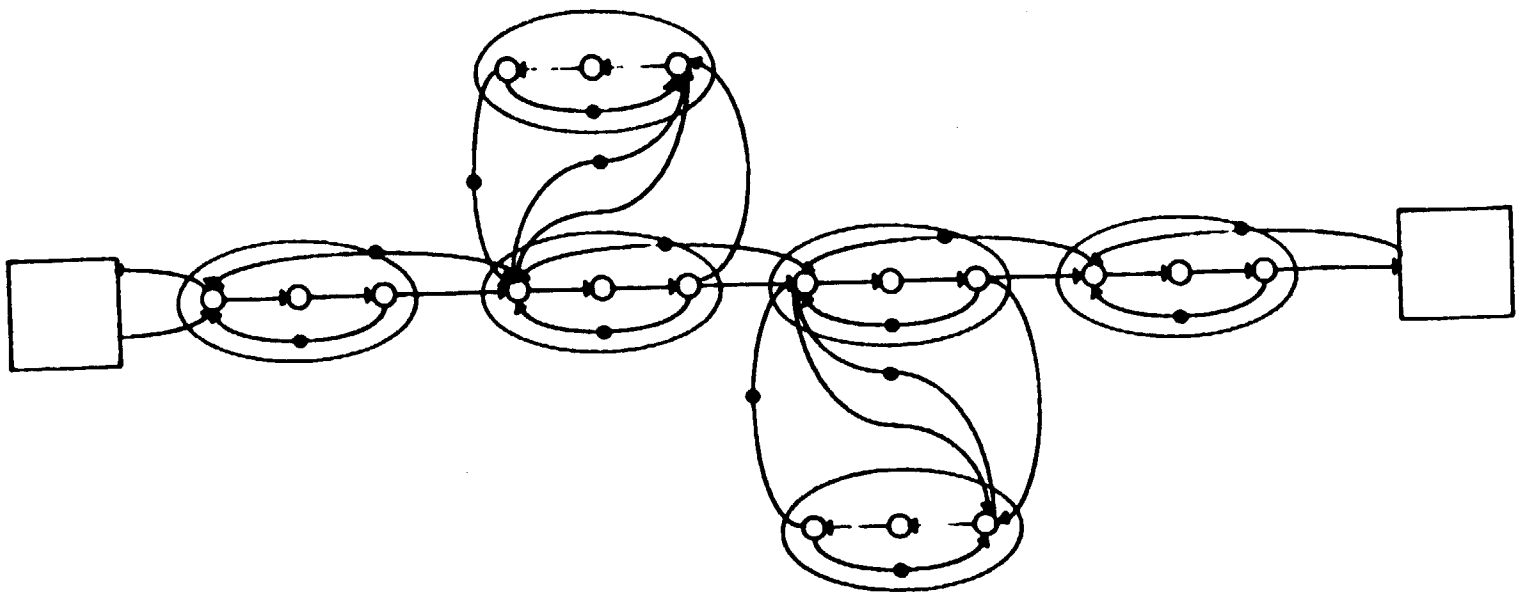


Figure 32. CMG of the graph in Figure 31 using Single Node Model.

```
# ***  Simulation of a graph with iterative loops  ***
# ***  It is simulated with full range of resources ***
# ***  and with two different priority assignments  ***

Graph Graph with iterative loops  TBO(GLB) = 960  TBIO(GLB) = 1600
    Nodes 6
    Sources 1
    Sinks 1
    Places 3
    Resources -1                # *** From 1 to 6 resources ***
    Priority 4 3 2 1 6 5        # Alternate assignment 5 6 1 2 3 4
    Tokens 1 7 9                # Data available at the input node
                                #  and in outputs of the iterative loops
    Model Single
    Input 1                     # The input node is node 1
    Output 4                    # The output node is node 4
    Times              # Global time assignment
        Read    70              # These time assignments are for all
        Process 210             # nodes in the graph. They can be
        Write   40              # overridden later on.

    Node 1
    Inputs 1
    Outputs 2

    Node 2
    Inputs 2 7
    Outputs 3 6

    Node 3
    Inputs 3 9
    Outputs 4 8
```

Figure 33. Graph Description File for the second experiment.

```
Time                    # Local time assignment
    Read   140          # These time assignments override
    Process 420         # the global time assignments
    Write   60          #

Node 4
Inputs 4
Outputs 5

Node 5
Inputs 6
Outputs 7
Time                    # Local time assignment
    Read   140          # These time assignments override
    Process 420         # the global time assignments
    Write   90          #

Node 6
Inputs 8
Outputs 9

Source 1
Outputs 1
Time
    Write 890           # Source output write time is TBD(LB) - IN1
                        # Write time = 960 - 70

Sink 2
Inputs 5
Time
    Read 70

End                     # This ends the Graph Description File
```

Figure 33. (Continuation).

APPENDIX A

THE ATAMM PROCEDURE MODEL FOR CURRENT PROCESSING OF LARGE GRAINED CONTROL AND SIGNAL PROCESSING ALGORITHMS

Presented at

National Aerospace and Electronics Conference
Dayton, Ohio

May 1988

THE ATAMM PROCEDURE MODEL FOR CONCURRENT PROCESSING OF LARGE GRAINED CONTROL AND SIGNAL PROCESSING ALGORITHMS

John W. Stoughton and Roland R. Mielke
Department of Electrical and Computer Engineering
Old Dominion University

ABSTRACT

An overview is presented of a model for describing data and control flow associated with the execution of large grained, decision free algorithms in a special distributed computer environment. Identified by the acronym ATAMM, which represents Algorithm–To–Architecture Mapping Model, the model provides a basis for relating an algorithm to its execution in a dataflow multicomputer environment. The ATAMM model features a marked graph Petri net description of the algorithm behavior with regard to both data and control flow. The model provides an analytical basis for calculating performance bounds on input characteristics which are demonstrated in the paper.

INTRODUCTION

The development of new computer architectures based upon distributed, multiprocessor organizations [1,2] is motivated mainly by the requirement for increased speed and greater throughput capability in complex signal processing applications [3]. With the advent of high–density microelectronics the construction of parallel architectures consisting of identical, special purpose computing elements is now a reality [4],[5]. A number of models for describing the behavior of algorithms in this setting have been developed [6]–[8]. However, these models represent only the data flow and do not adequately display the complex issues of communication and control flow which must occur in any realization. Thus, it has been difficult to investigate how to effectively match the decomposition and scheduling of algorithms to the structure and control of parallel architectures. The importance of better understanding the relationship between algorithms and architectures is only now becoming recognized [9].

This paper presents an overview of a graph theoretic model for describing both data and control flow associated with the concurrent processing of large grained algorithms in a special distributed computer environment. This model is identified by the acronym ATAMM which represents Algorithm To Architecture Mapping Model.

The purpose of the ATAMM model is important for three reasons. First, the model provides a hardware independent context in which to investigate the relative merits of different algorithm decomposition and implementation strategies. Second, the model defines the data flow and control flow which must be manifested by any dataflow computer architecture implementing the decomposed algorithm. Third, the model provides an analytical basis for performance evaluation.

The problem domain of the ATAMM model consists of large–grained, decision–free algorithms with computationally complex primitive operations which are assumed to be implemented in a dedicated distributed dataflow environment. The algorithms are such as may be found in (but not limited to) large scale signal processing and control applications. A potential multicomputer environment might consist of two to twenty processing elements composed of VHSIC technology.

ATAMM MODEL DEVELOPMENT

The composition of the algorithms of interest may be such that two or more operations can be performed concurrently. Thus, the potential exists for decreasing the computational time required to executing the algorithm by increasing the computational resources which process the large grained primitive operations.

The hardware environment (Figure 1) for executing the decomposed algorithms is assumed to consist of R identical processors or functional units (FUNs) where R has a value in the range of two to twenty. This range of resources is suggested for practical reasons due to the large–grained aspect of the algorithm decomposition and the need to maintain small communication times relative to process times. Each FUN is a processor having local memory for program storage and temporary input and output data containers. Each FUN can execute any algorithm primitive operation. The FUNs share a common global memory (GLM) which may be either centralized or distributed. The coordination of FUNs in relation to data and control flow is directed by the graph manager (GRM). The GRM also may be centralized or distributed. Transaction rules provide that output created by the completion of a primitive operation is placed into global memory only after the output data containers have been emptied. That is, outputs must be consumed as inputs to successor primitive operations before allowing new data to fill the output locations. Assignment of a functional unit to a specific algorithm primitive operation is made by the GRM only when all inputs required by the operation are available in global memory and a functional unit is available.

The algorithm to be executed has its data flow represented in a directed graph termed the algorithm directed graph (ADG). The ADG provides a description of the operand data flow and operation sequence required by the algorithm decomposition. Vertices of the ADG are in a one–to–one correspondence with each occurrence of a primitive operation. The ADG contains an edge (i,j) directed from vertex i to vertex j if the output of primitive operation i is an input operand for primitive operation j. When constructing an algorithm graph,

ices or nodes (primitive operations) are displayed as es. and edges (input–output signals) are displayed as :ted line segments connecting appropriate vertices. 'ces and sinks for input and output signals are esented as squares. Sources from constants are not ılly included in the algorithm graph: however, ıgles are used for this purpose when necessary.

To illustrate, consider the computations for the ₂ equation

$$x(k) = Ax(k-1) + Bu(k)$$

output equation

$$y(k) = Cx(k)$$

re x is a p–vector, u is an m–vector, y is an r–vector, A and B are constant matrices. The primitive ations are defined as matrix multiplication and or addition. The algorithm directed graph for this rithm decomposition is shown in Figure 2. Note that ı edge is labeled with the corresponding operands and nodes are labeled to indicate the associated putational operation.

Petri–nets have been established as an approp– ₂ model for describing systems defined by some ıence of events. Without argument, the algorithm cted graph satisfies this general aspect. Further, e computers need to communicate and be controlled the occurrence of certain events, the Petri–net ›mes a suitable theoretical vehicle for the ATAMM lel. Certain physical characteristics of the class of ›lems under consideration lead to a simplified Petri– representation. (For a formal description of Petri– features, the reader is referred to references [10–12].)

Considering the data flow in an algorithm directed ›h, the execution of a primitive operation is precon– ›ned on the availability of input signals (or ands). This process may be directly modeled by a i–net "transition" which is "enabled" for "firing" n input "places" to the transition are marked with .ens". Because the signal or data availability is a .ry condition, it is appropriate that the tokens are ted to the set {0,1} in order to associate places ditions) to transactions (events) in a binary way. A i–net having such restricted input and output tions is called an ordinary Petri–net. The rpretation of places in the system model developed is the availability of a signal. That is, the absence token indicates the absence of a data signal, and the ence of a token indicates the availability of a data al. Petri–nets having such restricted markings are d safe or one–bounded Petri–nets. Finally, the mption is made that the algorithms under considera– contain no conflict or decision making such as then–else" or "do–while" statements, thus limiting Petri–net places to having one input transition and output transition. This class of restricted Petri–nets alled marked graphs. Therefore, the Petri–nets used his report are ordinary, safe marked graphs.

Limiting the model for consideration of decision– algorithms is made because the resulting marked ›h models are better understood than general ri–nets and hold the potential for the development of formance bounds for concurrent processing strategies.

An algorithm marked graph (AMG) is a marked ›h which represents a specific algorithm decomposi– ı and is identical in topology to the corresponding ›rithm directed graph. The AMG represents the first component in the development of the ATAMM model. The construction rules and symbols are the same as the ADG except that the edges are marked with tokens to represent the availability of data. That is, edge (i,j) is marked with a token if an output from primitive operator i is available as an input to primitive operator j. The presence of a token on an edge is indicated by a solid dot placed on the edge. The vertices correspond to transitions which may fire after being enabled by the availability of all input data tokens. The decomposed state equation represented in Figure 2 is also used to illustrate the AMG. It should be noted that the initial conditions for the recursion are represented by tokens on the loop edges.

The AMG is a useful tool for representing decom– posed algorithms and for displaying data flow within an algorithm. However, the AMG does not identify proce– dures that a computing structure must manifest in order to perform the computing task.

Algorithm requirements and the computing environment may now be integrated into a comprehen– sive Petri–net model to complete the ATAMM model. The model consists of a Petri–net marked graph called the computational marked graph (CMG). The CMG displays the data flow and control flow required to implement a decomposed algorithm in a multiprocessor data flow computer architecture. Before defining this model, it is helpful to define an intermediate graph called the node marked graph (NMG), [13].

A NMG is a Petri–net representation of the computing behavior of a FUN for each AMG operation. Three primary activities, reading of input data from global memory, processing of input data to compute an output, and writing of output data to global memory, are represented as transitions (vertices) in the NMG. Data and control flow paths are represented as places (edges), and the presence of signals is notated by tokens marking appropriate edges. The conditions for firing the process and write transitions of the NMG are as defined for a general Petri–net, while the read transition has one additional condition for firing. In addition to having a token present on each incoming signal edge, a functional unit must be available for assignment to the primitive operation before the read node can fire. Once assigned, the functional unit is used to implement the read, process, and write operations before being returned to a queue of available FUNs.

The NMG of interest in this paper requires control signals indicating that empty data containers are available to receive new output as input edges to the read transition. Therefore, initiation of the node operation requires not only the availability of input data and a functional unit, but also the availability of empty output data containers in global memory. This model is shown in Figure 4.

A computational marked graph (CMG) is constructed from a particular AMG and the NMG according to the following rules.

1. Source and sink nodes in the AMG are repre– sented by source and sink nodes in the CMG.

2. Nodes corresponding to primitive operations in the AMG are represented by NMGs in the CMG.

3. Edges in the AMG are represented by edge pairs, one forward directed for data flow and one backward directed for control flow, in the CMG.

The play of the CMG proceeds according to the following graph rules.

1. A node is enabled when all incoming edges are marked with a token. An enabled node fires by encum–

bering one token from each incoming edge, delaying for some specified transition time, and then depositing one token on each outgoing edge.

2. A source node and a sink node fire when enabled without regard for the availability of a FUN.

3. A node operation is initiated when the read node of an NMG is enabled and when a FUN is available for assignment to the NMG and thus fires the read node. A FUN remains assigned to an NMG until completion of the firing of the write node of the NMG. Supervision of this logical assignment of the FUN is managed by the GRM.

For illustration, the CMG corresponding to the algorithm graph of Figure 2 is shown in Figure 4. The CMG is useful because it clearly displays the data and control flow which must occur in any hardware implementation of the model process, and because it provides a hardware independent context in which to evaluate process performance.

The ATAMM model consists of the algorithm marked graph, the node marked graph, and the computational marked graph, and the data flow architecture. A pictorial description of the ATAMM model is shown in Figure 5.

ATAMM MODEL GRAPH CHARACTERISTICS

The theoretical analysis of the CMG from the standpoint of marked graph theory is beyond the scope of this paper and may be found in [14]. However, several properties are noted below for clarity.

Let the CMG be a marked graph consisting of m places and n transitions. The m–vector M_k is the marking vector resulting from the firing of some sequence of k transitions. It may be shown that the number of tokens contained in any directed circuit of the CMG is invariant under transition firings.

The CMG is live for all appropriate initial marking vectors. That is, for a marking M if, for all markings reachable from M, it is possible to fire any transition of the CMG by progressing through some transition firing sequence.

The CMG is said to be consistent. That is, there exists a marking M and a transition firing sequence from M back to M such that every transition occurs at least once. In addition, each transition of G occurs an equal number of times in a firing sequence from a marking M back to M.

The CMG is said to be safe for marking M if, for all markings reachable from M, no place contains more than one token.

PERFORMANCE MEASURES

In this section, performance measures indicating computing speed and throughput capacity are defined. Bounds for these quantities are calculated analytically from the AMG and CMG. This information is essential for efficiently matching algorithm decompositions with architecture implementations. The work presented in this section is extension of recent investigations of the performance of Petri–nets [15],[16] and marked graphs [17].

Assume that R FUNs are available for the algorithm execution. A computational task is initiated when an input data token from the source node is encumbered. Task output occurs when a corresponding output data token is deposited at the output sink node. A task is completed when all computing associated with the task is completed. However, task output and task completion do not always coincide as may be found in iterative signal processing algorithms in which initial conditions for the next iteration may occur after an output has been calculated. Task completion is usually indicated in the AMG or the CMG by the return of the graph to some steady–state initial marking. To facilitate measurement of throughput capacity, it is assumed that tasks are initiated periodically with new input data sets. New data sets are available continuously as input tokens from the input source node. Included in this problem class are iterative algorithms where the present task requires as inputs data from previous task calculations.

Concurrency, at any instant, falls into one of two categories. On one hand, different functional units (FUNs) may be performing simultaneously several primitive operations belonging to a particular task within in the graph. This type of concurrency is referred to as vertical concurrency and has a direct effect on task computing speed. It is limited by the number of primitive operations that can be performed simultaneously in a given algorithm decomposition, and by the number of FUNs available. The second type of concurrency relates to FUNs which may be operating on different input tasks within the graph. This type of concurrency has a direct effect on throughput capacity. It is limited by the capacity of the graph to accommodate additional task inputs, and by the number of functional units available to implement the tasks. In the following it is shown that the process of algorithm decomposition imposes bounds on the amount of vertical concurrency and horizontal concurrency possible in a given problem. If sufficient computing resources are available, operation at these bounds can be achieved. If the number of computing resources is limited, the bounds can not be reached simultaneously and trade–offs between the amount of vertical concurrency and horizontal concurrency are possible.

Three performance measures for concurrent processing are defined. The performance measure TBIO is the computing time which elapses between a task input and the corresponding task output. The performance measure TT is the computing time which elapses between a task input and the completion of all computation associated with that task. The performance measure TBO is the computing time which elapses between successive task outputs when the graph is operating periodically in steady–state. The first two parameters, TBIO and TT, are indicators of computing speed and thus reflect the degree of vertical concurrency. The third parameter, TBO, is a measure of throughput capacity and thus reflects the degree of horizontal concurrency, when compared to TT.

Lower bounds of these measures may now be outlined, and may be found in detail in [14]. Consider an AMG representing a decomposed algorithm. The lower bound for TBIO is the shortest time required for a data token from the data input source to propagate through the graph to the data output sink. Similarly, the lower bound for TT is the shortest time required to complete all computing activity initiated by the injection of a data token from the data input source. These shortest times are the actual performance times when only a single task is active in the graph during any time interval (no horizontal concurrency), and as many computing resources as are required are available (maximum vertical concurrency). Under these operating conditions, lower bounds for TBIO and TT are calculated by identifying

certain longest paths in a graph obtained from the algorithm marked graph. This new graph, called the modified algorithm marked graph, MAMG, is defined and then used to determine lower bounds for TBIO and TT.

The construction of the modified AMG proceeds by the following rules. Let p_i be a place of of the AMG, directed from transition t_r to transition t_s, which contains a token of the initial marking. Then the MAGM may be obtained from the original AMG by

1. Deleting place p_i from the AMG;

2. Adding place p_{i1}, directed from the data input source to transition t_s, is added to G;

3. Adding a new output sink s_i different from all other output sinks, and a new place p_{i2}, directed from transition t_r to s_i; and

4. Repeating 1–3 for each place of the AMG containing a token of the initial marking.

Let P_i be the ith directed path in the MAMG from the data input source to the output sink. The lower bound for TBIO is defined as

$$TBIO_{LB} = Max \{ T(P_i) \},$$

where the maximum is taken over all paths P_i in the MAGM and $T(P_i)$ denote the sum of transition times for transitions contained in P_i.

Let P_i be the ith directed path in the MAMG from the data input source to any output sink. The lower bound for TT is defined as

$$TT_{LB} = Max \{ T(P_i) \}$$

where $T(P_i)$ denote the sum of transition times of transitions contained in P_i, and the maximum is taken over all paths P_i in the MAGM.

To illustrate, $TBIO_{LB}$ and TT_{LB} are computed for the AMG shown in Figure 2 for which the following transition times are assumed: T(1)=4, T(2)=1, T(3)=5, and T(4)=6. The MAMG is shown in Figure 6. It may be easily shown that $TBIO_{LB}$=10 and TT_{LB}=11.

A lower bound for the performance measure TBO is now determined from the CMG representing a decomposed algorithm. It is assumed that operating conditions are set to maximize horizontal concurrency. That is, data tokens are continuously available at the data input source, and as many computing resources as needed can be called to perform primitive operations. With these conditions, the graph plays periodically in steady–state, and TBO_{LB} is the shortest time possible between successive outputs. Let C_i be the ith directed circuit in the CMG. The notation $T(C_i)$ denotes the sum of transition times of transitions contained in C_i, and $M(C_i)$ denotes the number of tokens contained in C_i. Then,

$$TBO_{LB} = Max \{ T(C_i) / M(C_i) \},$$

where the maximum is taken over all directed circuits in the CMG.

The CMG in Figure 4 has many directed circuits. However, the directed circuit which contains all NMG nodes of transitions 2 and 4 contains only one token and maximizes the ratio $T(C_i) / M(C_i)$. Therefore, the shortest time possible between successive outputs in this graph is TBO_{LB}=7.

STRATEGY FOR OPTIMUM TIME PERFORMANCE

Of interest is the development of an operating strategy for the ATAMM model which achieves optimum time performance with a minimum number of computing resources. Unfortunately, this problem is equivalent to a class of scheduling problems which is known to be NP–complete. Thus, there exists no algorithm for obtaining an optimum solution which is better than enumerating all possible solutions and then choosing the best one. However, a suboptimal operating strategy which achieves optimum time performance, but possibly requires more than the minimum number of computing resources, has been developed and is illustrated in this section.

When presented with continuously available input data sets, the natural behavior of a data flow architecture results in operation where new data sets are accepted as rapidly as the available resources permit. That is, the architecture naturally operates at high levels of horizontal concurrency with the possible loss of capability for achieving high levels of vertical concurrency. This results in performance characterized by high throughput rates, $TBO=TBO_{LB}$, but relatively poor task computing speed so that $TBIO>>TBIO_{LB}$ and $TT>>TT_{LB}$. In many signal processing and control applications, it is important to achieve both high throughput rate and high task computing speeds. The suboptimal operating strategy presented in this section results in performance having the following characteristics.

1. When $R>R_{Max}$, operation achieves $TBIO_{LB}$, TT_{LB}, and TBO_{LB}. R_{Max} is computed in implementing the strategy, and represents the minimum number of resources which insures maximum horizontal concurrency and maximum vertical concurrency under this strategy.

2. When $R_{Max}>R>R_{Min}$, operation achieves $TBIO_{LB}$ and TT_{LB}, but $TBO>TBO_{LB}$. The strategy preserves task computing speed or vertical concurrency at the expense of throughput rate or horizontal concurrency. R_{Min} is also computed in implementing the strategy, and represents the minimum number of resources needed to maintain vertical concurrency with limited horizontal concurrency.

3. The rate at which new data is presented to the CMG must be limited. This is accomplished by adding a control transition connected in a directed circuit with the data input source. The control transition imposes a minimum delay of D time units between inputs. Delay D is chosen according to the following rule:

$$D = \begin{cases} TBOLB & R > R_{Max} \\ TBO_{Min} & R_{Max} > R > R_{Min} \\ TCE & R_{Min} > R > 1. \end{cases}$$

TCE denotes the total computing effort required to complete the task, and TBO_{Min}, R_{Max}, and R_{Min} are computed as part of the operating strategy design procedure.

The operating strategy design process consists of five steps. These steps are presented and explained in the remainder of this section. An operating strategy is developed for the example algorithm graph shown in Figure 7 to illustrate each step as it is presented.

Step 1. Choose a convenient transition firing rule. For the example algorithm graph, the highest to lowest priority ordering of the transitions is chosen as (5,4,3,7,2,6,1).

Step 2. Determine TBO_{LB}. The CMG corresponding to the example algorithm graph is shown in Figure 8. The directed circuit identified in this figure contains 6 transition time units and 2 tokens, and maximizes the ratio $T(C_i)/M(C_i)$ for all directed circuits. Therefore, $TBO_{LB}=3$.

Step 3. Determine the resource utilization envelope of a single task required for maximum vertical concurrency at steady-state with $TBO = TBO_{LB}$ under the assumption of unlimited resources. The play of the example algorithm graph under these conditions is shown in Figure 9, and the resulting resource utilization envelope is shown in Figure 10.

Step 4. Stabilize the resource utilization envelope by adding control places as necessary. If the time between inputs to the CMG is increased above TBO_{LB}, the resource utilization envelope may change from that observed in Step 3. Since knowledge of the envelope is required to calculate the number of required resources, additional places are appended to the AMG and the CMG to freeze the shape of the envelope. For example, the play of the example algorithm graph of Figure 7 with an injection time of 4 is shown in Figure 11. At this slower injection rate, transition 6 fires one time unit earlier. To prevent time movement of transition 6, a control place directed from transition 2 to transition 6 is added. This place prevents the firing of transition 6 until transition 2 has completed firing. Thus the resource utilization envelope computed for an input period of TBO_{LB} is the envelope for all input periods $TBO>TBO_{LB}$.

Step 5. Compute R_{Max}, R_{Min}, and $TBO_{Min}(R)$ using the resource utilization envelope. R_{Max} is determined by overlaying resource utilization requirements, each delayed by TBO_{LB} with respect to the previous one, as shown in Figure 12 for the example. R_{Max} is equal to the largest resource requirement during any time interval within the steady state operating period. R_{Min} is the minimum number of resources necessary to insure maximum vertical concurrency with no horizontal concurrency. This number is equal to the maximum resource requirement indicated in the resource utilization envelope for a single task. For the example problem, $R_{Max}=5$ and $R_{Min}=3$. The value of TBO_{Min} for each resource number R between R_{Max} and R_{Min} inclusive, is determined by increasing the delay between overlapping resource utilization envelopes until the maximum resource requirement is R. TBO_{Min} is the smallest input delay to produce this resource requirement. For the example, the calculations of TBO_{Min} for R=3 are illustrated in Figure 13. The results of these calculations are $TBO_{Min}(3)=4$.

The performance degradation as a function of R of the example algorithm graph is summarized in Figure 14 which shows the thruput rate or performance margin as a function of R. Note that for the example, no improvement in thruput is available for $R>R_{Max}$.

CONCLUSION

The ATAMM model has been demonstrated to be a useful graph theoretic model for describing data and control flow associated with the execution of large grained, decision free algorithms in a special distributed computer environment. The ATAMM model has been shown to provide an analytical basis for calculating performance bounds on thruput characteristics and suboptimum performance behavior. The ATAMM model leads directly to the communcation and data flow specifications for a data flow architecture and thus becomes the basis of design for these structures.

ACKNOWLEDGEMENT

The paper is based on research work which was supported in part by NASA Langley Research Center under Grant NAG-1-683.

REFERENCES

1. P. Treleaven, D. Brownbridge and R. Hopkins. "Data-driven and demand-driven computer architecture." Computing Surveys, vol. 14, pp. 93-143, March 1982.

2. V. Srini, "An architectural comparison of dataflow systems," Computer, pp. 68-88, March 1986.

3. W. Rheinbolt, "Report of the panel on future directions in computational mathematics, algorithms, and scientific software," sponsored by NSF Grant DMS-85-3483, SIAM, 1985.

4. T. Longo, G. Herzog and D. Maxwell, "A fast single chip 1750A CPU and compatible support components in VHSIC-size CMOS technology," Proceedings of the Government Microcircuit Applications Conference, pp. 317-320, 1986.

5. W. Wehner, W. Everhart, S. Shankar and K. Stalsberg, "A VSHIC architecture for highly parallel image understanding," Proceedings of the Government Microcircuit Applications.

6. M. Sowa and T. Murata, "A data flow computer architecture with program and token memories," IEEE Transactions on Computers, vol. 31, pp.820-821, September 1982.

7. K. Kavi, B. Buckles and U. Narayan Bhat, "A formal definition of data flow graph models," IEEE Transactions on Computers, vol. 35, pp. 940-945, November 1986.

8. M. Granski, I. Koren and G. Silberman, "The effect of operation scheduling on the performance of a data flow computer," IEEE Transactions on Computers, vol. 36, pp. 1019-1029, September 1987.

9. L. Jamieson, H. Siegel, E. Delp and A. Whinston, "The mapping of parallel algorithms to reconfigurable parallel architectures, "Proceedings of Future Directions in Computer Architecture and Software, D. Agrawal Ed., ARO Contract DAAG29–81–D–0100, pp. 147–154, May 1986.

10. J. Peterson, Petri–net Theory and the Modeling of Systems, Englewood Cliffs, N.J.: Prentice–Hall, 1981.

11. T. Murata, "Circuit theoretic analysis and synthesis of marked graphs," IEEE Transactions on Circuits and Systems, vol. 24, pp. 400–405, July 1977.

12. T. Murata, "Modeling and analysis of concurrent systems,"Handbook of Software Engineering, C. Vick and C. Ramamoorthy Editors, pp. 39–63, Van Nostrand Reinhold, 1984.

13. J. Stoughton and R. Mielke, "Petri net model for concurrent processing of complex algorithms," Proceedings of the Government Microcircuit Applications Conference, pp. 11–14, November 1986.

14. R. Mielke, J. Stoughton, and S. Som, "Modeling and performance bounds for concurrent processing," 8th International Conference on Distributed Computing Systems, San Jose CA, June 13–17, 1988.

15. J. Sifakis, "Performance evaluation of systems using nets," Net Theory and Applications, W. Brauer Editor, pp. 307–319, Springer–Verlag, 1979.

16. C. Ramamoorthy and G. Ho, "Performance evaluation of asynchronous concurrent systems using Petri–nets," IEEE Transactions on Software Engineering, vol. 6, pp. 440–449, September 1980.

17. T. Murata, "Synthesis of decision–free concurrent systems for prescribed resources and performance," IEEE Transactions on Software Engineering, vol. 6, pp. 525–530, November 1980.

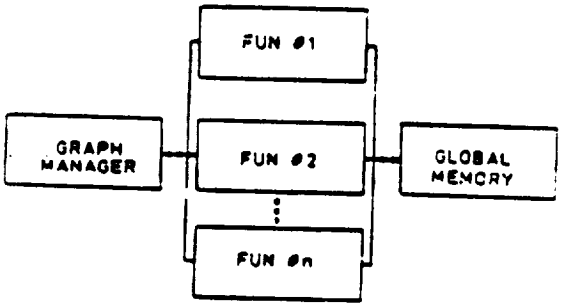


Figure 1. Representative ATAMM Architecture

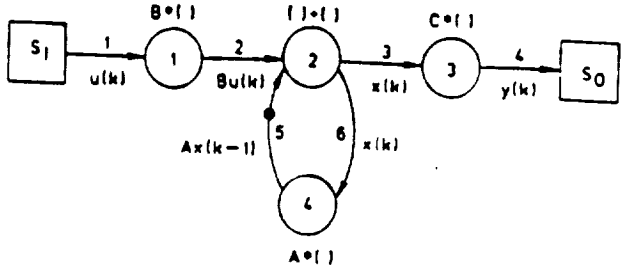
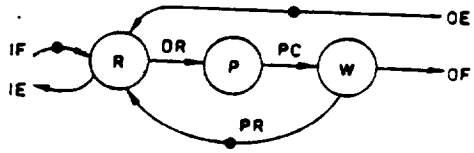


Figure 2. Algorithm Marked Graph – Example 1



NMG EDGE LABELS

IF	Input Buffer Full
IE	Input Buffer Empty
DR	Data Read
PC	Process Complete
PR	Process Ready
OE	Output Buffer Empty
OF	Output Buffer Full

Figure 3. ATAMM Node Marked Graph Model

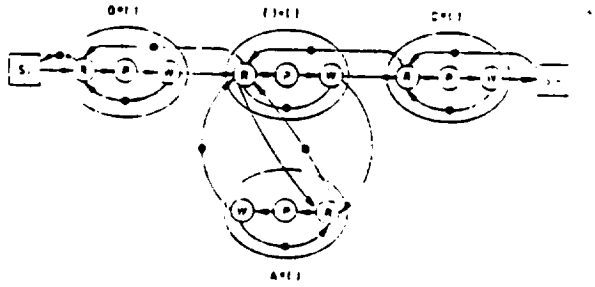


Figure 4. Computational Marked Graph – Example 1

ORIGINAL PAGE IS
OF POOR QUALITY

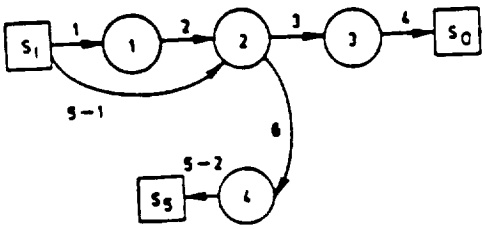


Figure 6. Modified AMG – Example 1

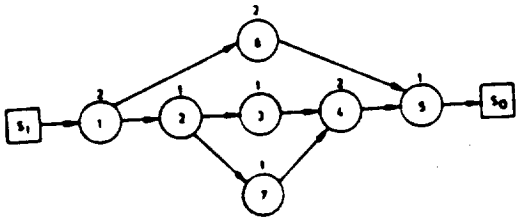


Figure 7. Algorithm Marked Graph – Example 2

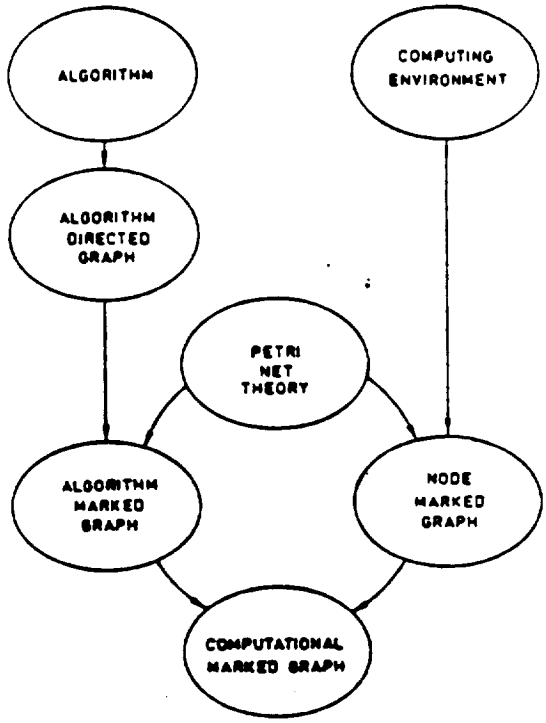


Figure 5. ATAMM Model Components

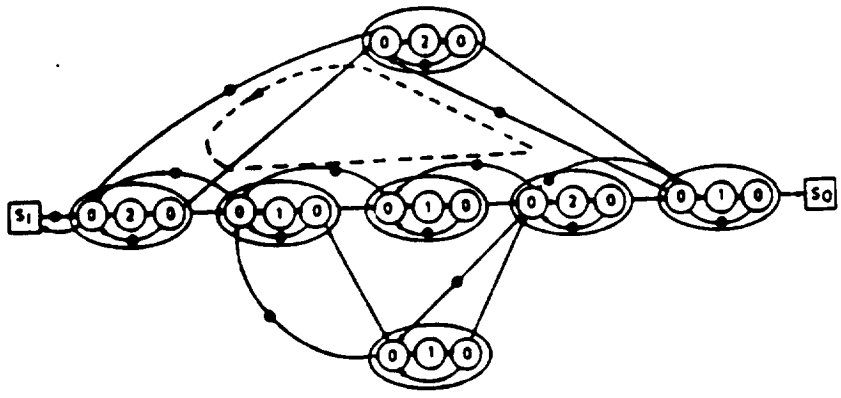


Figure 8. Computational Marked Graph – Example 2

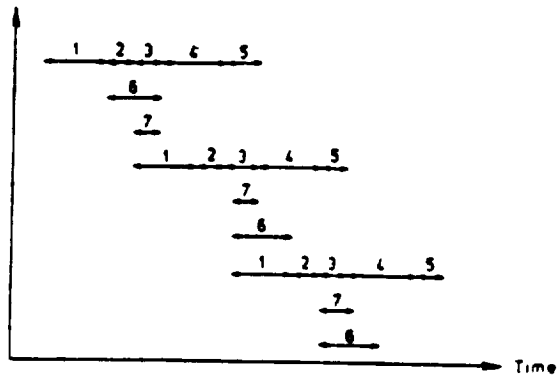


Figure 9. Graph Play With TBO=3 and Unlimited Functional Units

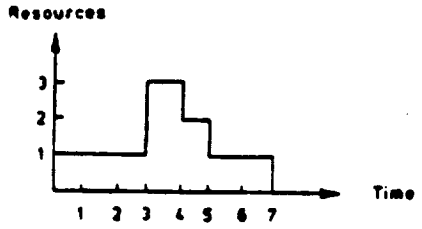


Figure 10. Resource Utilization Envelope – Example 2

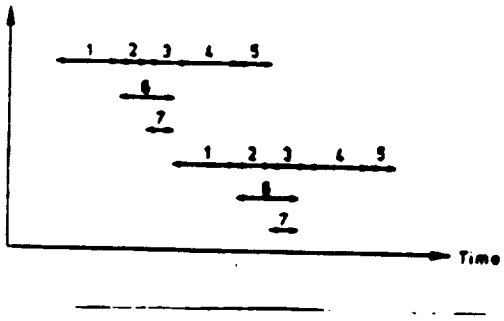


Figure 11. Graph Play With TBO=4 W/O Control Edges

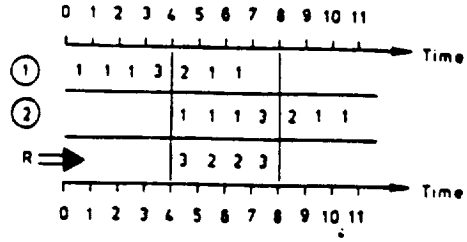


Figure 12. Resource Envelope Overlay Diagram – TBO=3.0

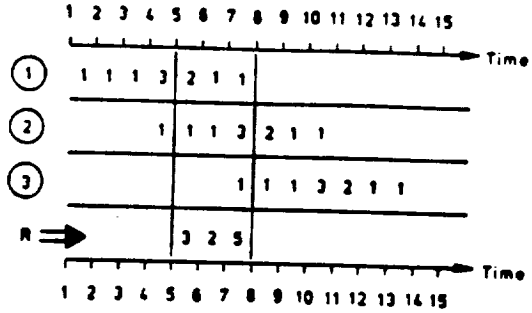


Figure 13. Resource Envelope Overlay Diagram – TBO=4.0

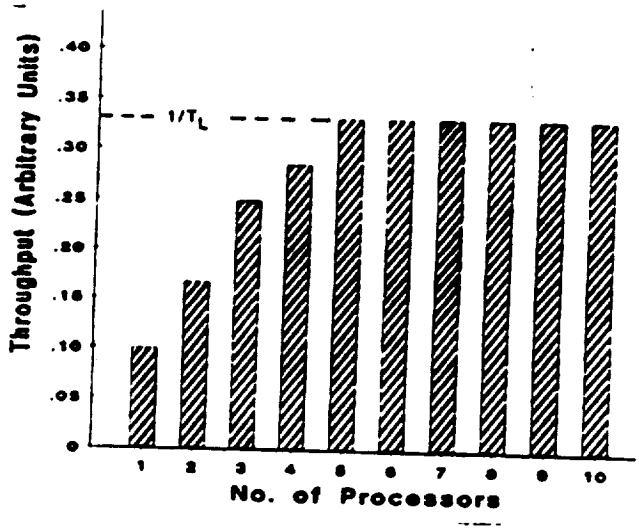


Figure 14. Performance Margin – Example 2

APPENDIX B

MODELING AND PERFORMANCE BOUNDS FOR CONCURRENT PROCESSING

Presented at

International Conference on Distributed Computing Systems
San Jose, California

June 1988

MODELING AND PERFORMANCE BOUNDS FOR CONCURRENT PROCESSING

Roland R. Mielke, John W. Stoughton and Sukhamoy Som
Department of Electrical and Computer Engineering
Old Dominion University
Norfolk, Virginia

ABSTRACT

The development of a new graph theoretic model for describing the relation between a decomposed algorithm and its execution in a multiprocessor environment is presented. Called ATAMM, the model consists of a set of Petri net marked graphs which incorporates the general specifications of a data flow architecture. The model is useful for representing decision–free algorithms having large–grained, computationally complex primitive operations. Performance measures of computing speed and throughput capacity are defined. The ATAMM model is used to develop analytically lower bounds for these quantities.

I. INTRODUCTION

The development of a new graph theoretic model for describing data and control flow associated with the execution of large–grained algorithms in a special distributed computing environment is presented. The model is identified by the acronym ATAMM which represents Algorithm To Architecture Mapping Model. The purpose of such a model is to provide a basis for establishing rules for relating an algorithm to its execution in a multiprocessor environment. Specifications derived from the model lead directly to the description of a data flow architecture. The availability of the ATAMM model is important for at least three reasons. First, it provides a context in which to investigate algorithm decomposition strategies without the need to specify a specific computer architecture. Second, the model identifies the data flow and control dialog required of any data flow architecture which implements the algorithm. And third, the model provides a basis for calculating analytically performance bounds for computing speed and throughput capacity.

The problem domain of the ATAMM model consists of decision free algorithms with computationally complex primitive operations which are assumed to be implemented in a dedicated data flow environment. The algorithms are such as may be found in (but not limited to) large scale signal processing and control applications. The anticipated multiprocessor environment is assumed to consist of two to twenty processing elements for concurrent execution of the various algorithm primitives.

The development of new computer architectures based upon distributed, multiprocessor organizations [1], [2] is motivated mainly by the requirement for increased speed and greater throughput capability in complex signal processing applications [3]. Recent advances in the production of high–density microelectronics [4] has made possible the construction of parallel architectures consisting of identical, special purpose computing elements [5]. A number of models for describing the behavior of algorithms in this setting have been developed [6]–[8]. However, these models represent only the data flow and do not adequately display the complex issues of communication and control flow which must occur in any realization of the model. For this reason, it has been difficult to investigate how to effectively match the decomposition and scheduling of algorithms to the structure and control of parallel architectures. The importance of better understanding the relationship between algorithms and architectures is only now becoming recognized [9].

In Section II of the paper, the modeling process to describe algorithms in data flow architectures, ATAMM, is presented. The model consists of three Petri net marked graphs called the algorithm marked graph (AMG), the node marked graph (NMG), and the computational marked graph (CMG). In Section III, time performance measures for concurrent processing are defined. The ATAMM model is used as the basis for calculating analytically lower bounds for these performance measures. An example is presented to illustrate these concepts, and the results of experimental runs on actual multiprocessor hardware are reported.

II. ATAMM MODEL DEVELOPMENT

In this section the ATAMM model to describe concurrent processing of decomposed algorithms is presented. The model consists of a set of Petri net marked graphs which incorporate general specifications of communication and processing associated with each computational event in a data flow architecture. First, a detailed description of the problem context is stated. This is followed by the definition of the ATAMM model consisting of the algorithm marked graph, the node marked graph, and the computational marked graph. Some familiarity with Petri nets [10] and marked graphs [11] is assumed in this presentation.

The problems of interest are decision–free, computationally complex problems as are often found in signal processing and control applications. A problem description normally results in the definition of a function given by the triple (X,Y,F). The set X represents the set of admissible inputs, the set Y represents the set of admissible outputs, and F:X–>Y is the rule of correspondence which unambiguously assigns exactly one element from Y to each element of X. Associated with a computational problem is one or more algorithms. An algorithm is an explicit mathematical statement, expressed as an ordered set of primitive operations, which explains

CH2541-1/88/0000/0538$01.00 © 1988 IEEE

how to implement the rule of correspondence F. In general, a given problem can be decomposed by several different primitive operator sets. Also, for a given primitive operator set, there are often different orderings of primitive operations which can be specified to carry out the problem. Of special interest are algorithm decompositions in which two or more primitive operations can be performed concurrently. For such decompositions, the potential exists for decreasing the computational time required to solve the problem by increasing the computational resources which implement the primitive operations.

The hardware environment for executing the decomposed algorithms is assumed to consist of R identical processors or functional units (FUNs) where R has a value in the range of two to twenty. This range of resources is suggested for practical reasons due to the large-grained aspect of the alogrithm decomposition and the need to maintain small communication times relative to process times. Each FUN is a processor having local memory for program storage and temporary input and output data containers. Each FUN can execute any algorithm primitive operation. The FUNs share a common global memory (GLM) which may be either centralized or distributed. The coordination of FUNs in relaton to data and control flow is directed by the graph manager (GRM). The GRM also may be centralized or distributed. Output created by the completion of a primitive operation is placed into global memory only after the output data containers have been emptied. That is, outputs must be consumed as inputs to successor primitive operations before allowing new data to fill the output locations. Assignment of a functional unit to a specific algorithm primitive operation is made by the GRM only when all inputs required by the operation are available in global memory and a functional unit is available.

An algorithm marked graph is a marked graph which represents a specific algorithm decomposition. Vertices of the algorithm graph are in a one-to-one correspondence with each occurrence of a primitive operation. The algorithm graph contains an edge (i,j) directed from vertex i to vertex j if the output of primitive operation i is an input for primitive operation j. Edge (i,j) is marked with a token if an output from primitive operator i is available as an input to primitive operator j. When constructing an algorithm graph, vertices (primitive operations) are displayed as circles, and edges (input-output signals) are displayed as directed line segments connecting appropriate vertices. The presence of a token on an edge is indicated by a solid dot placed on the edge. Source transitions and sink transitions for input and output signals are represented as squares. Sources for constants are not usually included in the algorithm marked graph: however, triangles are used for this purpose when necessary.

To illustrate the construction of an algorithm marked graph, consider the problem of computing the output of a discrete linear system given a sequence of inputs to the system. Let the system be described by the state equation

$$x(k) = Ax(k-1) + Bu(k)$$

and output equation

$$y(k) = Cx(k),$$

where x is a p-vector, u is an m-vector, and y is an r-vector. The primitive operations are defined as matrix multiplication and vector addition, and the natural algorithm decomposition resulting from the state equation description is selected. The algorithm marked graph for this decomposed algorithm is shown in Figure 1. The initial marking indicates that initial condition data are available.

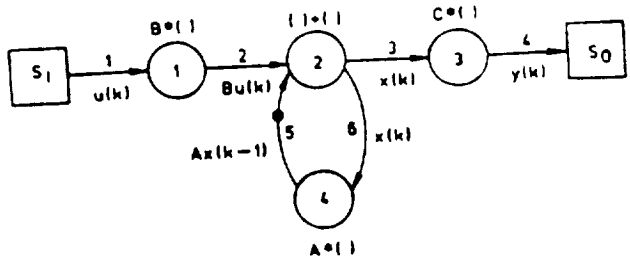
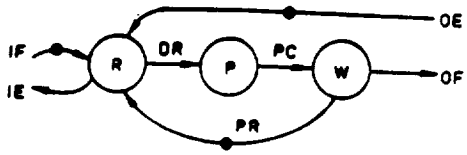


Figure 1. Algorithm marked graph for discrete system equation.

The algorithm marked graph is a useful tool for representing decomposed algorithms and for displaying data flow within an algorithm. However, the algorithm graph does not display procedures that a computing structure must manifest in order to perform the computing task. In addition the issues of control, time performance, and resource management are not apparent in this graph. These important aspects of concurrent processing are included in the ATAMM model through the definition of two additional graphs. The node marked graph is defined to model the execution of a primitive operation. The computational marked graph, obtained from the AMG and the NMG by a set of construction rules, integrates both the algorithm requirements and the computing environment requirements into a comprehensive graph model. These additional marked graphs are defined in the following.

A node marked graph is a Petri net representation of the performance of a primitive operation by a functional unit. Three primary activities, reading of input data from global memory, processing of input data to compute output data, and writing of output data to global memory, are represented as transactions (vertices) in the NMG. Data and control flow paths are represented as places (edges), and the presence of signals is notated by tokens marking appropriate edges. The conditions for firing the process and write transitions of the NMG are as defined for a general Petri net, while the read transition has one additional condition for firing. In addition to having a token present on each incoming signal edge, a functional unit must be available for assignment to the primitive operation before the read node can fire. Once assigned, the functional unit is used to implement the read, process, and write operations before being returned to a queue of available FUNs. The initial marking for an NMG consists of a single token in the "process ready" place. The NMG model is shown in Figure 2.



NMG EDGE LABELS

IF	Input Buffer Full
IE	Input Buffer Empty
DR	Data Read
PC	Process Complete
PR	Process Ready
OE	Output Buffer Empty
OF	Output Buffer Full

Figure 2. ATAMM node marked graph model.

A computational marked graph(CMG) is constructed from the AMG and the NMG by the following rules.

1. Source and sink nodes in the algorithm marked graph are represented by source and sink nodes in the CMG.

2. Nodes corresponding to primitive operations in the algorithm marked graph are represented by NMGs in the CMG.

3. Edges in the algorithm marked graph are represented by edge pairs, one forward directed for data flow and one backward directed for control flow, in the CMG. The initial marking for the edge pair consists of a single token in the forward–directed place if data are available, or a single token in the backward–directed place if data are not available.

The play of the CMG proceeds according to the following graph rules.

1) A node is enabled when all incoming edges are marked with a token. An enabled node fires by encumbering one token from each incoming edge, delaying for some specified transition time, and then depositing one token on each outgoing edge.

2) A source node and a sink node fire when enabled without regard for the availability of a FUN.

3) A primitive operation is initiated when the read node of an NMG is enabled and a FUN is available for assignment to the NMG. A FUN remains assigned to an NMG until completion of the firing of the write node of the NMG.

In order to illustrate the construction of a computational marked graph, the CMG corresponding to the algorithm marked graph of Figure 1 is shown in Figure 3. The computational marked graph is useful because it clearly displays the data and control flow which must occur in any hardware implementaion of the model process, and because it provides a hardware independent context in which to evaluate process performance.

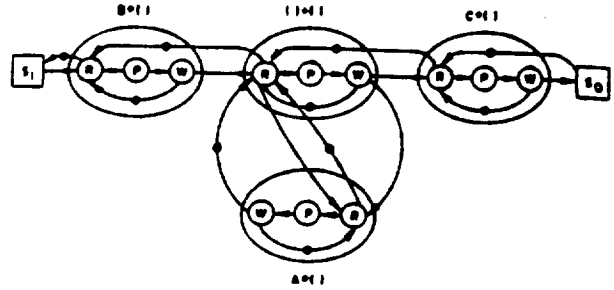


Figure 3. ATAMM computational marked graph model for discrete system equation.

The complete ATAMM model consists of the algorithm marked graph, the node marked graph, and the computational marked graph. A pictoral display of this model is shown in Figure 4. In the next section, time performance characteristics of the ATAMM model are investigated.

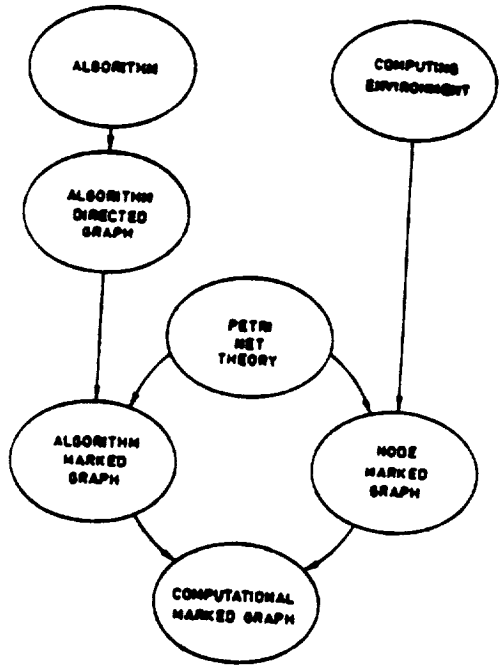


Figure 4. ATAMM model components.

III. PERFORMANCE BOUNDS

The importance of the ATAMM model is that it establishes a context in which to investigate the performance of decomposed algorithms in multiprocessor data flow architectures. In this section, performance measures indicating computing speed and throughput capacity are defined. Bounds for these quantities are calculated analytically from the algorithm marked graph and the computational marked graph. This information is

essential for efficiently matching algorithm decompositions with architecture implementations. The work presented in this section is an interesting application and extension of recent investigations of the performance of Petri nets [12], [13] and marked graphs [14].

It is assumed that a decomposed algorithm is implemented in a multiprocessor architecture containing R computing resources or functional units. Each functional unit is capable of performing any of the primitive operations whose sequence defines the decomposition. A computational task is initiated when an input data token from the source node is encumbered. Task output occurs when a corresponding output data token is deposited at the output sink node. A task is completed when all computing associated with the task is completed. It should be noted that task output and task completion do not always coincide. In many iterative signal processing algorithms, computing to generate initial conditions for the next iteration often occurs after an output has been calculated. Task completion is usually indicated in the AMG or the CMG by the return of the graph to some steady–state initial marking. To facilitate measurement of throughput capacity, it is assumed that tasks are repeated periodically with new input data sets. New data sets are available continuously as input tokens from the input source node. Included in this problem class are iterative algorithms where the present task requires as inputs data from previous task calculations.

Concurrency in this problem setting occurs in two ways. First, different functional units may perform simultaneously several primitive operations belonging to a single task. This type of concurrency is referred to as vertical concurrency. Vertical concurrency has a direct effect on task computing speed. It is limited by the number of primitive operations that can be performed simultaneously in a given algorithm decomposition, and by the number of functional units available to perform the primitive operations. Second, different functional units may perform simultaneously primitive operations belonging to different tasks sequentially input to the computing system. Called horizontal concurrency, this type of concurrency has a direct effect on throughput capacity. It is limited by the capacity of the graph to accommodate additional task inputs, and by the number of functional units available to implement the tasks. In the following it is shown that the process of algorithm decomposition imposes bounds on the amount of vertical concurrency and horizontal concurrency possible in a given problem. If sufficient computing resources are available, operation at these bounds can be achieved. If the number of computing resources is limited, the bounds cannot be reached simultaneously and trade–offs between the amount of vertical concurrency and horizontal concurrency are possible.

Three performance measures for concurrent processing are defined. The first two parameters, TBIO and TT, are indicators of computing speed and thus reflect the degree of vertical concurrency. The third parameter, TBO, is a measure of throughput capacity and thus reflects the degree of horizontal concurrency.

Definition 1: TBIO. The performance measure TBIO is the computing time which elapses between a task input and the corresponding task output.

Definition 2: TT. The performance measure TT is the computing time which elapses between a task input and the completion of all computation associated with that task.

Definition 3: TBO. The performance measure TBO is the computing time which elapses between successive task outputs when the graph is operating periodically in steady–state.

The remainder of this section is devoted to developing lower bounds for these performance measures.

Let G denote an algorithm marked graph representing a decomposed algorithm. The lower bound for TBIO is the shortest time required for a data token from the data input source to propagate through the graph to the data output sink. Similarly, the lower bound for TT is the shortest time required to complete all computing activity initiated by the injection of a data token from the data input source. These shortest times are the actual performance times when only a single task is active in the graph during any time interval (no horizontal concurrency), and as many computing resources as are required are available (maximum vertical concurrency). Under these operating conditions, lower bounds for TBIO and TT are calculated by identifying certain longest paths in a graph obtained from the algorithm marked graph. This new graph, called the modified algorithm graph G_M, is defined and then used to determine lower bounds for TBIO and TT.

Definition 4: Modified Algorithm Graph. Let p_i be a place of G, directed from transition t_r to transition t_s, which contains a token of the initial marking. The modified algorithm graph G_M is obtained from the graph G by the following construction rules.

1. Place p_i is deleted from G.

2. A new place p_{i1}, directed from the data input source to transition t_s, is added to G.

3. A new output sink s_i different from all other output sinks, and a new place p_{i2}, directed from transition t_r to s_i, are added to G.

4. The above rules are repeated for each place of G containing a token of the initial marking.

Lower bounds for TBIO and TT are presented in Theorem 1 and Theorem 2 respectively.

Theorem 1: Lower Bound for TBIO. Let P_i be the ith directed path in G_M from the data input source to the data output sink, and let $T(P_i)$ denote the sum of transition times for transitions contained in P_i. Then,

$$TBIO_{LB} = Max \{ T(P_i) \},$$

where the maximum is taken over all paths P_i in graph G_M.

Proof. Without loss of generality, let t_f be the last transition in all paths P_i directed from the data input source to the data output sink. Transition t_f is enabled when each input place for t_f contains a token. Since by assumption a computing resource is available, t_f fires as soon as it becomes enabled. Let p_q be the last input place for t_f to acquire a token, and let t_g be the input transition for place p_q. Continuing this labeling procedure results in a backward path construction process. This process is repeated, first at t_g, and then at each succeeding transition until the data input source is reached, identifying a path P_j. By the construction process for the path, it is clear that $T(P_j) = \text{Max}\ \{ T(P_i) \}$, where the maximum is over all paths P_i in G_M. It is also clear that $TBIO_{LB}$ can be no shorter than $T(P_j)$ so that $TBIO_{LB} \geq T(P_j)$. Since a computing resource is available when each transition in P_j is enabled, the time between input and corresponding output can be no longer than $T(P_j)$ so that $TBIO_{LB} \leq T(P_j)$. Therefore, $TBIO_{LB} = T(P_j) = \text{Max}\ \{ T(P_i) \}$, where the maximum is over all paths P_i in G_M. This completes the proof.

Theorem 2: Lower Bound for TT. Let P_i be the ith directed path in G_M from the data input source to any output sink, and let $T(P_i)$ denote the sum of transition times of transitions contained in P_i. Then,

$$TT_{LB} = \text{Max}\ \{ T(P_i) \}$$

where the maximum is taken over all paths P_i in graph G_M.

Proof. By the construction rules for graph G_M, a task is initiated when input data tokens are input from the data input source, and is completed when all output sinks have accepted tokens. Therefore, TT is the time which elapses from injection of input tokens to the arrival of a token at the last fired output sink. Let $T(P_t) = \text{Max}\{T(P_i)\}$, P_i in G_M, be the longest path time of paths from the data input source s_I to any output sink, say s_t. Since a token must reach sink s_t before a task is completed, it follows that $TT_{LB} \geq T(P_t)$. Since a resource is available for each transition to fire when enabled, and since P_t is the longest path in G_M, it also follows that $TT_{LB} \leq T(P_t)$. Therefore, $TT_{LB} = T(P_t) = \text{Max}\{T(P_i)\}$, where the maximum is over all paths P_i in G_M. This completes the proof.

To illustrate the application of Theorem 1 and Theorem 2, $TBIO_{LB}$ and TT_{LB} are computed for the algorithm graph shown in Figure 1. For this example, the following transition times are assumed: $T(1) = 4$, $T(2) = 1$, $T(3) = 5$, and $T(4) = 6$. The modified algorithm graph corresponding to Figure 1 is shown in Figure 5. The modified algorithm graph contains two paths directed from the data input source s_I to the data output sink s_O. Path P_1 consists of edge set {1, 2, 3, 4} with $T(P_1) = 10$, and path P_2 consists of edge set {5–1, 3, 4} with $T(P_2) = 6$. Therefore, since $T(P_1) > T(P_2)$, path P_1 determines the lower bound for TBIO and $TBIO_{LB} = 10$. The modified algorithm graph contains two additional directed paths from the data input source s_I to the output sink s_5. Path P_3 consists of edge set {1, 2, 6, 5–2} with $T(P_3) = 11$, and path P_4 consists of edge set {5–1, 6, 5–2} with $T(P_4) = 7$. Since $T(P_3) > T(P_1) > T(P_4) > T(P_2)$, path P_3 determines the lower bound for TT and $TT_{LB} = 11$.

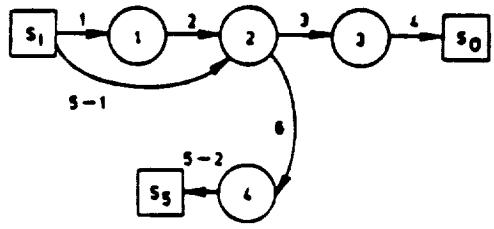


Figure 5. Modified algorithm graph for Figure 1.

Next a lower bound for the performance measure TBO is presented. Let G be a computational marked graph representing a decomposed algorithm. It is assumed that operating conditions for G are set to maximize horizontal concurrency. That is, data tokens are continuously available at the data input source, and as many computing resources as needed can be called to perform primitive operations. With these conditions, the graph plays periodically in steady-state, and TBO_{LB} is the shortest time possible between successive outputs.

Theorem 3: Lower Bound for TBO. Let G be a computational marked graph and let C_i be the ith directed circuit in G. The notation $T(C_i)$ denotes the sum of transition times of transitions contained in C_i, and $M(C_i)$ denotes the number of tokens contained in C_i. Then,

$$TBO_{LB} = \text{Max}\ \{ T(C_i) / M(C_i) \},$$

where the maximum is taken over all directed circuits in G.

Proof. Without loss of generality, let t_f be the output transition in G so that an output is produced each time t_f

completes firing. Then TBO_{LB} is the minimum firing period of transition t_f. It is shown in [15] (pp. 58–60) that the minimum firing period of each transition of a marked graph is given by $Max\{T(C_i)/M(C_i)\}$, where the maximum is taken over all directed circuits C_i in G. Therefore, the theorem follows.

The computational marked graph shown in Figure 3 is used to illustrate Theorem 3. This CMG contains many directed circuits. However, the directed circuit which contains all NMG nodes of transitions 2 and 4 contains only one token and maximizes the ratio $T(C_i) / M(C_i)$. Therefore, the shortest time possible between successive outputs in this graph is $TBO_{LB} = 7$.

The optimum time performance for this example algorithm is described by the following characteristics. The algorithm accepts an input and issues an output every 7 time units. Each input requires a total of 11 time units of processing, and an output is issued 10 time units after the input is accepted. It can be shown by simulation that 3 functional units are required to achieve this performance. The addition of more functional units will not improve the computing speed or throughput rate for this algorithm decomposition.

IV. CONCLUSION

A new model useful for understanding the relationship between decomposed algorithms and data flow architectures has been presented. Named ATAMM for Algorithm To Architecture Mapping Model, the model consists of Petri net marked graphs called the algorithm marked graph, the node marked graph, and the computational marked graph. Time performance measures of time between input and output (TBIO), task time (TT), and time between outputs (TBO) were defined. Then lower bounds for the performance measures were calculated analytically from the modified algorithm graph and the computational marked graph. An example to illustrate these concepts was presented.

Simulation tools and an actual hardware prototype have been developed to test and validate the ATAMM model. The simulation software package [16] consists of a PC–based computer model of the CMG. Algorithms are entered to the package by specifying the algorithm marked graph, and simulation output consists of a graphical display of the movement of tokens. An accompanying diagnostic software package [17] automatically computes and displays performance measures and other performance data. A hardware prototype [18] has also been constructed to validate the ATAMM operating rules and to provide a benchmark for testing the simulation software. The architecture is shown in Figure 6 and is one of several candidates which could be used to perform concurrent operations according to the ATAMM rules. A primary motivation for this particular design was the availability of hardware. The system consists of S–100 crates having an Intel 8088 CPU card, multiple serial I/O channels, and 32K memory. An IBM/XT is used to host the system and to down load algorithm graph descriptions to the system. A number of decomposed algorithms, including those presented here, have been investigated using these tools.

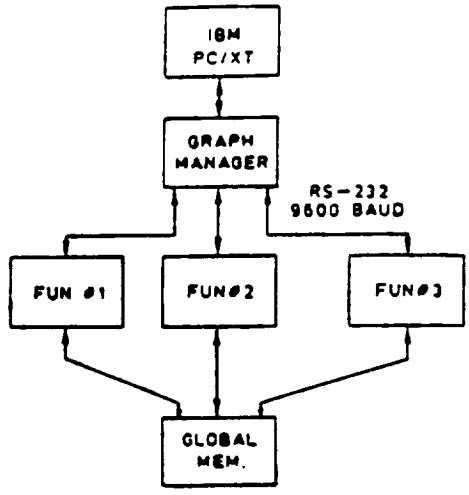


Figure 6 Prototype hardware configuration for ATAMM validation.

Continuing research is designed to generalize the ATAMM model and is focused in three main areas. The present model assumes that all functional units are identical and that each is able to perform all primitive operations. An important extension is to model the situation where there are two or more different groupings of processors where each group is able to perform only a subset of the required primitive operations. The present model represents only decision–free algorithms. Another important extension is to develop the capability to admit algorithms containing data–dependent branching points. Finally, methods for achieving optimum time performance are being studied in the context of the ATAMM model.

ACKNOWLEDGEMENT

The work reported here was supported in part by the NASA Langley Research Center under Grant NAG–1–683.

REFERENCES

[1] P. Treleaven, D. Brownbridge and R. Hopkins, "Data–driven and demand–driven computer architecture," Computing Surveys, Vol. 14, pp. 93 – 143, March 1982.

[2] V. Srini, "An architectural comparison of dataflow systems," Computer, pp. 68 – 88, March 1986.

[3] W. Rheinbolt, "Report of the panel on future directions in computational mathematics, algorithms, and scientific software," sponsored by NSF Grant DMS–85–3483, SIAM, 1985.

[4] T. Longo, G. Herzog and D. Maxwell, "A fast single chip 1750A CPU and compatible support components in VHSIC–size CMOS technology," Proceedings of the Government Microcircuit Applications Conference, pp. 317 – 320, 1986.

[5] W. Wehner, W. Everhart, S. Shankar and K. Stalsberg, "A VSHIC architecture for highly parallel image understanding," Proceedings of the Government Microcircuit Applications Conference, pp. 117 – 120, November 1986.

[6] M. Sowa and T. Murata, "A data flow computer architecture with program and token memories," IEEE Transactions on Computers, Vol. 31, pp.820 – 824, September 1982.

[7] K. Kavi, B. Buckles and U. Narayan Bhat, "A formal definition of data flow graph models," IEEE Transactions on Computers, Vol. 35, pp. 940 – 948, November 1986.

[8] M. Granski, I. Koren and G. Silberman, "The effect of operation scheduling on the performance of a data flow computer," IEEE Transactions on Computers, Vol. 36, pp. 1019 – 1029, September 1987.

[9] L. Jamieson, H. Siegel, E. Delp and A. Whinston, "The mapping of parallel algorithms to reconfigurable parallel architectures," Proceedings of Future Directions in Computer Architecture and Software, D. Agrawal Ed., ARO Contract DAAG29–81–D–0100, pp. 147 – 154,May 1986.

[10] J. Peterson, Petri Net Theory and the Modeling of Systems, Englewood Cliffs, N.J.: Prentice–Hall, 1981.

[11] T. Murata, "Circuit theoretic analysis and synthesis of marked graphs," IEEE Transactions on Circuits and Systems, Vol. 24, pp. 400 – 405, July 1977.

[12] J. Sifakis, "Performance evaluation of systems using nets," Net Theory and Applications, W. Brauer Editor, pp. 307 – 319, Springer–Verlag, 1979.

[13] C. Ramamoorthy and G. Ho, "Performance evaluation of asynchronous concurrent systems using Petri nets," IEEE Transactions on Software Engineering, Vol. 6, pp. 440 – 449, September 1980.

[14] T. Murata, "Synthesis of decision–free concurrent systems for prescribed resources and performance," IEEE Transactions on Software Engineering, Vol. 6, pp. 525 – 530, November 1980.

[15] T. Murata, "Modeling and analysis of concurrent systems," Handbook of Software Engineering, C. Vick and C. Ramamoorthy, Editors, pp. 39 – 63, Van Nostrand Reinhold, 1984.

[16] K. Jackson, R. Tymchyshyn, R. Mielke and J. Stoughton, "Simulation software for concurrent processing," Proceedings of the IEEE Southeastcon Conference, pp. 82 – 86, April 1987.

[17] R. Obando, "Simulation software for performance evaluation of concurrent processing," Master's Thesis, Old Dominion University, Norfolk, Virginia, October 1987.

[18] J. Stoughton and R. Mielke, "Petri net model for concurrent processing of complex algorithms," Proceedings of the Government Microcircuit Applications Conference, pp. 11 – 14, November 1986.